VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Android Board Game Implementation

Implementace deskové hry pro Android

2015 Ľubomír Sokolovský

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Bachelor Thesis Assignment

Student: **Ľubomír Sokolovský**

Study Programme: B2647 Information and Communication Technology

Study Branch: 2612R025 Computer Science and Technology

Title: Implementace deskové hry pro Android
Android Board Game Implementation

Description:

The goal of the bachelor thesis is design and implementation of a multiplayer board game for OS Android platform. The game will be inspired by strategy board games (e.g. "Pandemic"). The application will support cooperative multiplayer networking against the computer. User interface of the game will be designed for tablets and smart phones, with simplified graphics.

1. Research of Android board games.
2. Design of the new board game and its set of rules.
3. Implementation of the game logic and user interface.
4. Design of the network communication protocol and implementation of the game server.
5. Application tests on the available devices.
6. Summary of the results.

References:

[1] James Steele, Nelson To, The Android Developer's Cookbook: Building Applications with the Android SDK, Addison-Wesley Professional, 2010, ISBN-13: 978-0321741233
[2] Reto Meier, Professional Android 4 Application Development, Wrox, 2012, ISBN-13: 978-1118102275
[3] Sayed Hashimi, Pro Android 2, Apress, 2010, ISBN-13: 978-1430226598

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **Mgr. Ing. Michal Krumnikl, Ph.D.**

Date of issue: 01.09.2013
Date of submission: 07.05.2015

doc. Dr. Ing. Eduard Sojka
Head of Department

prof. RNDr. Václav Snášel, CSc.
Dean of Faculty

I agree with publishing of this thesis as required by *Study and Examination Regulations for bachelor programs of VŠB-TUO* article 26, paragraph 9.

In Ostrava 7th May 2015 .....

I declare that I have accomplished this thesis individually. I have referenced all the literature and publications from which I have taken information.

In Ostrava 7th May 2015 .....

There is a lot of people that helped me in completion of this thesis. I would like to thank at least some of them. Ing. Martin Gajdičiar for the original topic idea. Ing. Peter Chovanec for help with server administration. Mgr. Radka Švaňová for grammar and vocabulary revision. All the testers that helped to make Eridian better. However, the most important thanks goes to Mgr. Ing. Michal Krumnikl PhD., my thesis supervisor. He is one of the few people who seem to know the answer to every question.

Abstrakt

Tahle práce se zaměřuje na vytvoření sady pravidel pro deskovou hru a jeji implementaci v aplikaci pro Android. Samotná implementace byla realizována pomocí .NET technologií - Xamarin for Android a SignalR. Úsilí vyústilo ve funkční deskovou hru pro Android s možným budoucím rozšířením pro zařízení iOS a Windows Phone. V konečném důsledku, práce rozšiřuje původní zaměření o zkoumáním možností multiplatformového vývoje a komunikačního modelu PUSH.

Klíčová slova: desková hra, Android, Mono, Xamarin, SignalR, PUSH

Abstract

This thesis focuses on creating a set rules for a board game and implementing them in an Android application. The implementation was realized using .NET technologies - Xamarin for Android and SignalR. The effort resulted in a working board game for Android with possible future extension for iOS and Windows Phone devices. Eventually, the thesis broadens the original focus by examining multiplatform development options and PUSH communication model.

Keywords: board game, Android, Mono, Xamarin, SignalR, PUSH

List of used abbreviations and symbols

2D	–	Two-dimensional
3D	–	Three-dimensional
AAA	–	(In game industry) High quality, high budget games, best-sellers
AL	–	Application Layer
API	–	Application Programming Interface
ART	–	Android Runtime
ASP	–	Active Server Pages, now part of ASP.NET
BL	–	Business Layer
CD	–	Compact Disc
CLR	–	Common Language Runtime
DL	–	Data Layer
DPI	–	Dots Per Inch
DTO	–	Data Transfer Object
EDGE	–	Enhanced Data Rates from GSM Evolution
GPRS	–	General Packet Radio Service
GPS	–	Global Positioning System
GSM	–	Global System for Mobile Communications
GUI	–	Graphical User Interface
Guid	–	Globally Unique Identifier
HSCSD	–	High-Speed Circuit-Switched Data
HSDPA	–	High-Speed Downlink Packet Access
HTML	–	Hypertext Markup Language
HTTP	–	Hypertext Transfer Protocol
HTTPS	–	Hypertext Transfer Protocol Secure
ID	–	Identifier
IDE	–	Integrated Development Environment
IIS	–	Internet Information Services
IP	–	Internet Protocol
IT	–	Information Technology
JSON	–	JavaScript Object Notation
LINQ	–	Language-Integrated Query
OS	–	Operating System

OWIN	–	Open Web Interface for .NET
PC	–	Personal Computer
PCL	–	Portable Class Library
Pub/Sub	–	Publish-Subscribe pattern
REST	–	Representational State Transfer
SDK	–	Software Development Kit
SOAP	–	Simple Object Access Protocol
SSL	–	Secure Sockets Layer
TCP	–	Transmission Control Protocol
UDP	–	User Datagram Protocol
UI	–	User Interface
UMTS	–	Universal Mobile Telecommunications System
URL	–	Uniform Resource Locator
UX	–	User Experience
VOIP	–	Voice over IP
WCF	–	Windows Communication Foundation
WP	–	Windows Phone
WSDL	–	Web Service Definition Language
XML	–	Extensible Markup Language

Contents

List of Figures

List of Code Samples

1 Introduction

Board games. Those tiny pieces of paper, plastic and wood that have the magic ability to bring together relatives and friends to share a small portion of their precious time and experience the feelings of joy and excitement (and frustration). Some of them are quite new, others have been with us for decades and some - like chess - have endured for centuries.

However, playing a board game requires many criteria to be met. Conditions that are difficult to fulfil nowadays. You have to be physically together with your co-players (or in-game-enemies). You have to own a giant box carrying all the small pieces. The set up takes often a lot of time. It is almost impossible to play a board game when travelling. And when you lose just one piece of the game equipment, you often have to buy the whole set again. Some board games have extremely extensive rules and a narrator or banker is needed. And then there are those shadowy players who are willing to give up fair play for a handful of tricks and an easy win.

Fortunately, modern technology allows us to overcome a lot of these limitations. Of course, they lack the authentic feel of honest old-school board games. But they balance the scales with many positives. It takes just a few short moments to start a game in your cellphone, tablet or computer. You can carry it everywhere - modern gadgets are both small and lightweight. The role of the narrator is taken by the system itself. Also, the program - if well designed and written - guarantees fair play for all players. Virtual reality offers a lot more interactivity and opens doors to whole new concepts of gameplay. And if you have access to the Internet, you don't have to be with the other players together. In fact, your game mates may be on different continents and speak entirely different languages.

This thesis guides the reader through the whole process of creating a game for mobile devices. First, it focuses on the initial inception, researching the design of a "perfect" game and setting up the rules of the final product. Later, when solid rules for the game have been established, the thesis describes the process of choosing a suitable platform and development tools. It discusses the possibilities and drawbacks of multi-platform mobile development. As the reader will later learn, said game will be designed for multiplayer cooperation, thus server and connection technologies will be discussed later.

After all that theoretical knowledge is gathered, the reader will be allowed to discover the process of implementation in detail, pinpointing some interesting observations and lessons learned. In the very end the thesis will offer the results of several tests, synthesizing them into a final verdict. There is no more time to waste. Let the game begin...

2 Motivation

The original trigger of the idea to implement a board game on smart devices was a game called Samarkand: Routes to Riches [1] created by David V. H. Peters and Harry Wu. Samarkand is a competitive multiplayer board game for 2 - 5 players. The game is set in the era of the greatest glory of Silk Road. Players can hire the representatives of various merchant families from different nations and beat their rivals.

The goal of Samarkand is to achieve as much points as possible before endgame criteria are met. The points are earned for gold coins, luxury resources and trade connections with other families. As soon as each family creates at least one trade connection, or at least one family creates 5 connections, the game ends and every player counts points score. There are more rules on how a family can take control over a certain region and how the points for luxury resources are evaluated. When combined, the decision-making processes during Samarkand gameplay become pretty complex. Although luck does play a small role in specifying the starting goals for each player, the game is won by thinking ahead, calculating every step and preventing your opponents from reaching their goals.

Although Samarkand may sound as a great game (and it definitely is), it suffers from several classic "issues" of board games. First of all, the initial game set up takes about 10 - 15 minutes. This is substantial amount of time given that an average gameplay lasts for 45 minutes.The board's dimensions are 56 x 56 cm. The game contains 20 plates of family representatives' portraits, 120 camel pawns, 50 trade connection cards, 66 resource cards, 4 score evaluation cards and 75 coins. This makes the game almost impossible to play when travelling. Players need enough space and a stable ground. Moreover, losing any of the parts makes the game unplayable.

As noted earlier, the score evaluation process is also quite complex. It is very easy to omit some resources or trade connection that should contribute to player's points. Last, but not least, a simple arithmetic error in the final calculation could also change the game's result. All those issues could be simply solved if Samarkand was a video game for portable devices, such as smartphones or tablets.

2.1 Rules of a Good Game

Of course, it would be very easy to simply take an existing game and implement its rules into the world of digital media. However, without the previous consent from the actual authors of Samarkand, this would mean a considerable infringement of their intellectual property. Moreover, copying an existing game is barely half the fun compared to creating a new one.

This idea has led the author of this thesis to search for the basic concepts that make a great game. The most common denominators and factors that make a good and attractive game are following [2] [3]:

[1] http://boardgamegeek.com/boardgame/66214/samarkand-routes-riches

[2] http://www.wikihow.com/Make-a-Game

[3] http://www.thegamesjournal.com/articles/GameTheory4.shtml

- the game should be easy to learn but difficult to master.
- there should be more than one way to win. And more than one way to lose.
- the game should be challenging, but achievable. Each risk should be motivated with appropriate reward.
- the amount of randomness and luck should be restricted, so that the game is played by players, not by chance.
- however, small amount of randomness makes the game more thrilling and adds to replayability value.
- other ways of ensuring good replayability of a game include varying objectives, different game setup, creating special abilities (or disabilities) and roles for players' characters, grouping players into teams, etc.
- games should be also thematically and graphically appealing, since the first impulse for choosing a game are often its visuals rather than rules and mechanisms
- it is desirable to make a unique and original game. However, the originality should not take price in clarity of the game rules. Some object and elements have a strong symbolic meaning that is intuitive to majority of players. They should not be altered for the sole purpose of uniqueness.

Of course, each of the sources provided even more advices and guidelines. Nonetheless, those above could be regarded as the basic conditions of a good game. The first iteration of game rules resulted in a concept called Eridian Wars. It loosely followed the principles of Samarkand. However, as the name suggests, direct conflicts were added based on the combat principles from an online mass multiplayer strategy, called Might and Magic: Heroes Kingdoms[4].

Yet, the concept seemed to be too mathematical and unappealing to players. Thus, new rules were added to Eridian Wars and a lot of old ones were changed. As a result, the game became too complicated, overly based on luck and generally unplayable.

2.2 Tabletop, EuroGames and Pandemic

Since theoretical knowledge seemed not enough, the author began to research and study the mechanics of actual board games. A great source of gameplay information is the Tabletop web video series [5]. Tabletop is a web show created by Wil Wheaton and Felicia Day. In each episode a board, card or role-playing game is presented in a short intro. After that a practical demonstration follows, as the host - Wil Wheaton - and his guests play the presented game.

[4] `http://mightandmagicheroeskingdoms.ubi.com/en.html`
[5] `http://geekandsundry.com/shows/tabletop/`

Many of the games features in TableTop had exciting rules and captivating gameplay. The most promising of them was group of games called, German-style games, or Eurogames [6]. Eurogames are strategic board games that rely more on players' skills than luck. Even if a Eurogame is competitive, all players remain in game until the end. Like Samarkand (but unlike The Eridian Wars), Eurogames do not present direct player conflicts or fights. Instead, they often feature indirect conflicts, like competition over rare resources. Eurogames are centered mainly on interesting game mechanisms and the theme is often secondary (and easily swappable with a different theme).

Over time, the author's attention was shifted from competitive games to cooperative. When human players unite together against the game mechanics itself, the game offers a lot more possibilities. Most of the unfair competition and balancing issues are removed. The game is also easier to learn, since new players can work together with more skilled ones. The most interesting (at least to the author of this thesis) of the Tabletop games was Pandemic.

Pandemic is a cooperative multiplayer board game for 2-4 players published in 2008 by Z-Man Games [7]. Designed by Matt Leacock, Pandemic portraits our world in a threat of 4 deadly diseases. Players take up the roles of various specialists with unique abilities and try their best to save the world. The only way to win Pandemic is to find cures for all diseases. However, there are several ways how to lose the game. More details about the game rules can be found in the official rule book.

Pandemic became so popular that it got a re-edition in 2013 and several expansions. As interesting as the game is, Pandemic became the subject of a study of distributed computational thinking among board game players [2]. In the experiment, three groups of students who had never played Pandemic before, were asked to play the game. As a result, the authors of the paper suggest broader usage of board games for learning computational skills outside computationally-based situations.

2.3 Eridian - Rules

Synthesizing all the acquired knowledge - both theoretical and practical - and following the great example of Pandemic, a final set of rules was established. The game of Eridian was born. Set in a pseudo-medieval fantasy world, Eridian is a cooperative multiplayer game for 2, 3 or 5 players. The base story is that evil races are summoning a dark deity to destroy the world. The ritual of summoning is performed by 2/3/5 summoners (respective to the number of players) and it lasts 30 days. Players take up the roles of defenders of Eridian. They have to discover ancient magical symbols - called glyphs - to get mystical powers and banish the evil summoners before the ritual is completed.

The game starts with the players located in their respective capital cities. Evil summoners and their bodyguards are placed randomly on the map. Also the glyphs are placed randomly. However, they remain invisible unless a player's pawn stands on the same location.

[6] `http://boardgamegeek.com/wiki/page/Eurogame`

[7] `http://zmangames.com/home.php`

Each player has four action points available in his turn. After he/she uses all his/her actions, or passes the token, the enemy armies will spread. Then the next player's turn starts, also with four action points. When all players have made their turns, a new day begins and the first player in row can perform actions again.

Players can spend their action points on several actions. They can move to adjacent location. They can teleport (move to a far location instantly) to their capital or their tower. Players can also build buildings. Each player can build one tower (serving as a teleportation destination) and one library (serving as a drop point for carried glyphs). Action points can be also spent on fighting enemy armies, picking-up or dropping glyphs.

If a glyph is guarded by enemies, the glyph becomes corrupted and it would harm its carrier. Therefore, a player has to destroy the armies guarding the glyph and then cleanse it. The process of cleansing a glyph will consume player's all remaining action points. After the glyph is cleansed, the player can pick it up and carry it to his capital, or his library, where the glyph can be dropped. A player can carry only one glyph at a time.

When all glyphs have been gathered, all players receive a mystical titan blessing. They can then travel to locations where summoners are placed and banish them. Similar to cleansing, also banishing consumes all remaining action points. To win the game, players have to banish all summoners (they do not have to defeat all enemy armies).

The game can be lost in three different manners:

- players do not manage to banish all summoners within 30 day limit
- any enemy army of the size 3 occupies a capital city
- any enemy nation places 20 or more armies on the map

Similar to Pandemic, in case of 4 armies of the same nation would stack up on the same location, a special event happens instead. This special event has always the form of a penalty lasting for one or several game days. These penalties include: diminishing the maximum action points by one for all players, increased number of spreading enemy armies, moving players to random locations, temporary invulnerability of enemy armies, corrupting uncorrupted glyphs, etc.

3 Mobile Games Market Research

Since the commercial introduction of mobile phones in mid 80's of 20th century, telephony has witnessed a rapid development of technologies. Cell phones became more powerful, yet smaller and lighter every year. Providing phone calls soon became just one of many functionalities of these devices. As smart phones were developed, handheld devices overtook many of the features provided by computers and notebooks. With the introduction of tablets, the gap between mobile platforms and traditional computers has become smaller than ever. It is expected that because of the popularity of tablets, whole series of netbooks (small, low-cost, performance-limited notebooks) will cease to exist. [8]

And this is just the beginning. It is expected that smartphones and tablets will continue in their relentless attack on market shares. In many countries, more than half of the population will own a smartphone in the near future. Already now, 73% of South Koreans own a smartphone [9]. Current expectations state that in 2017 more than one third of the world's population will own a smartphone [10]. That means there will be at least 2.5 billion smart devices worldwide (it is worth to note that some people own more than one smart device, thus the number may be considerably higher). In combination with the ever-faster and more available Internet, IT industry faces a future of immense possibilities. Imagine a third of the world's population, 2.5 billion people connected to each other anywhere and anytime.

The rise of handheld devices and the Internet have created a similar starting ground for indie developers and mobile-device-oriented companies as classic gaming companies had in the late 80's and early 90's. With limited performance, smartphones and tablets are not suitable for AAA game titles. On the contrary, they are open to lighter weight innovative games. Moreover, thanks to online markets (Google Play, App Store, Windows Phone Store) developers are free of the burden joined with finding a suitable (and willing) publisher.

The portion of gaming industry belonging to titles for smart devices is increasing every year [11] [12] [13]. Handheld devices, and especially smartphones, have one great advantage over computers and notebooks. They are often carried literally everywhere. This might be one of the reasons for an emerging pattern in people's gaming habits. Many smartphone and tablet users, who have never played video games before, start playing (mostly casual) games on their mobile devices. Another great feature of handheld devices

[8] http://www.pcper.com/news/General-Tech/ASUS-kills-Eee-PC-and-shrinks-Atom-market

[9] http://web.archive.org/web/20150208071435/http://etc-digital.org/digital-trends/mobile-devices/mobile-smartphones/

[10] http://www.emarketer.com/Article/Worldwide-Smartphone-Usage-Grow-25-2014/1010920

[11] http://venturebeat.com/2013/06/06/global-games-market-to-hit-86-1b-by-2016-as-mobile-charges-ahead/

[12] http://www.proelios.com/global-games-market-grows-6-to-70-4bn-in-2013-number-of-gamers-worldwide-will-rise-above-1-2-billion/

[13] http://www.newzoo.com/insights/global-games-market-will-reach-102-9-billion-2017-2/

are their unique controls. The concept of controlling by touch, gesture and position eliminates the need of external input devices and creates opportunities for creating unique gaming experience. [14] . And it is not just touching, sliding and scrolling. Accelerometers, gyroscopes, GPSs, cameras and other smartphone inputs allow developers to create brand new, unique and revolutionary game genres [15].

However, the reader should not get the image that games for smartphones and tablets are just casual and indie titles. Many large gaming companies are creating lighter versions or spin-offs of their main titles. Also indie games can be very challenging and targeted on the hardcore part of gaming audience.

3.1 Androids, Apples, BlackBerries and Windows(es)

One of the tough decisions every mobile developer has to make is - which platform should he/she target? The problem with mobile platforms is that an application written for one platform will not run (not even install) on a different platform. For example, Android applications are written in Java, use Eclipse or Android Studio as IDE and run on Dalvik virtual machine (or ART in more recent Android versions). iPhone applications are written in Objective-C, developed in XCode and run natively on iOS. Moreover, you need a Mac to even perform the build for your code. The last big player, Windows Phone (WP), has apps written in C#, main IDE is Visual Studio and the application runs on CLR inside the Windows Phone OS (or Windows OS as of Windows 10). Similarly to iPhone apps, you need to have a PC to compile a WP application (however, Windows OS can be legally installed on a Mac device).

If we cannot develop for all mobile operating systems the first impulse would be to develop for the platform with the largest market share. It is worthy to note that mobile platform shares vary widely across the world. In this thesis we will focus on data that are valid globally. The most used mobile platform worldwide is undoubtedly Android. The second place belongs to Apple's iOS devices, followed by Windows Phone platform and continualy decreasing share of BlackBerry operating system [16].

The Android platform has clearly the largest market share. Moreover, publishing an application on Google Play is the cheapest and easiest, compared to Apple Store or Windows Phone Store. Until recent policy change in the Microsoft company [17], Java and Android Studio were also the only free development tools from the three platforms.

As it seems, the decision is very easy. Developing for Android covers most of the market. It does not require any paid tools and submitting the application to Google Play is the cheapest and most straightforward solution. Let's make an Android app!

[14] http://www.mcvuk.com/news/read/why-is-mobile-gaming-such-a-success/0121737

[15] http://en.wikipedia.org/wiki/Augmented_reality

[16] http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[17] http://news.microsoft.com/2014/11/12/microsoft-takes-net-open-source-and-cross-platform-adds-new-development-capabilities-with-visual-studio-2015-net-2015-and-visual-studio-online/

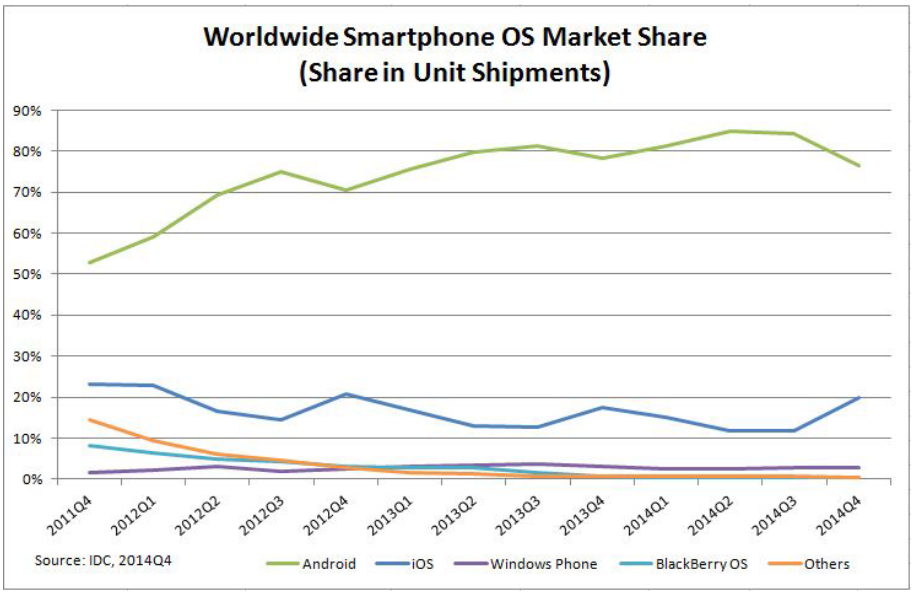

Figure 1: Smartphone OS market share development (IDC.com)

There is just one little catch. Although Android covers the largest portion of market, it does not produce the highest revenues [18] [19]. The highest total revenue is generated by iOS apps, while the highest relative revenue (with respect to number of application downloads) is created by Windows Phones. In fact, if you plan to release a paid application, you might hit serious demand issues. Android users are not usually used to pay for their applications [20]. And since developing for Android is so easy and available to almost everyone, there soon may be some copycats, producing a fake version of your application - for free.

One more factor that should not be omitted is the probable audience of our product - the Eridian game. Board games are usually played with friends and family (in contrast with hard-core competitive games, often played with random strangers). Users playing with their relatives or close friends regularely make in average 25-30% of the mobile-device gamers share [21].

To make a game accessible and appealing for whole families, it should have nice and attractive graphics and easily-understandable rules. But apart from design (of both graphics and game mechanics), can this audience specification help us in choosing the ideal mobile operating system? If families are our primary audience, we should look

[18] http://www.developereconomics.com/report/q1-2014-developer-revenue-model

[19] http://www.forbes.com/sites/tristanlouis/2013/08/10/how-much-do-average-apps-make/

[20] http://www.macobserver.com/tmo/article/new_report_shows_ios_users_spend_money_like_to_check_weather

[21] http://www.bigfishgames.com/blog/2014-global-gaming-stats-whos-playing-what-and-why/

for a platform that is used rather evenly by all age categories, with a possible peak in the 30-40-year-old users (parents of children and teenagers). When we compare several available studies [22] [23] [24], it is again the Android platform that fits those requirements most closely.

3.2 The Final Decision

So far, we have discovered that Android covers the largest part of market share. Its users' age distribution is fairly even, with a slight peak at 30-40-year-old. Developing for Android is also the cheapest possibility. Second best is iOS with the second largest market share and the highest absolute revenues. Windows Phone and BlackBerry are far behind in market share, but have significantly less saturated app markets. Yet, Windows Phone has the highest revenues per download.

The main inspiration for Eridian was Pandemic, which has an iOS version [25]. However, there are no Pandemic implementations for Android, BlackBerry or Windows Phone.

Now, let us consider the hardware available to the author of this thesis. The ownership of a Windows PC allows development of Android, BlackBerry and Windows Phone apps, while it is not sufficient for an iOS app development. All three available platforms provide emulators for application testing. However, the author has also an access to several physical Android devices, which gives another bonus point for this platform.

In the end, all those factors have led to the decision of implementing Eridian for Android (as the primary platform). Since the author intends to release the game for free, the fact that iPhone and iPad users are better payers does not play a role. A more important fact is that the author intends to implement the game for other platforms, later. And the programming language of his preference is C#. These factors will be crucial in the decision making process when choosing development tools - as we will see in the third part of this thesis.

[22] http://techcrunch.com/2012/09/26/forrester-iphone-app-users-young-and-wealthy-android-app-users-skew-older/

[23] http://www.iacquire.com/blog/smartphone-activities-study

[24] http://blogs.adobe.com/digitalmarketing/mobile/the-mobile-consumer-data-and-design-tips-to-consider/

[25] https://itunes.apple.com/us/app/pandemic-the-board-game/id700793523?mt=8

4 Client Side: Android on C#

In the first two parts of this thesis we have followed the process of creating a new board game and choosing the right platform for implementation. Although these processes were rather parallel with the client and server-side technical choices in the real implementation, this thesis will continue to describe those processes sequentially.

So far, the author has created a set of rules for Eridian - a cooperative multiplayer game inspired by a board game called Pandemic. The author has also researched the possibilities to implement his game for a mobile platform. His effort has led him to choosing the Android platform. Let us discuss this mobile operating system for a while.

4.1 The Java Tradition

Android is an open-source software stack that includes the operating system, middleware, and key mobile applications, along with a set of API libraries for writing applications that can shape the look, feel, and function of the devices on which they run. [6]

Android sits alongside a new wave of modern mobile operating systems designed to support application development on increasingly powerful mobile hardware.... Android is an ecosystem made up of a combination of three components:

- *A free, open-source operating system for embedded devices*
- *An open-source development platform for creating applications*
- *Devices, particularly mobile phones that run the Android operating system and the applications created for it [6]*

As many developers are aware, Android is tightly joined with Google's distribution of Java [8]. Being a free popular modern programming language that - thanks to a virtual machine - runs on every platform, Java was an obvious choice for Google. Although Google played with the idea to switch to a different programming language, because of a lawsuit with Oracle, they decided to stick with Java [26]. Java applications for Android can be developed in several IDEs. Formerly it was NetBeans and - mainly - Eclipse. However, since its release in May 2013, Android Studio has become the official development environment [27].

The build of the application can be performed on any machine as well - be it Windows PC, Mac or Linux. The major development drawback of Android platform is mainly its extremely slow emulator [28]. However, with the introduction of the Genymotion emulator, this issue can be avoided.

[26] http://appleinsider.com/articles/11/08/06/google_fighting_to_suppress_evidence_android_willfully_infringed_upon_oracles_java.html

[27] http://developer.android.com/tools/studio/index.html

[28] http://jolicode.com/blog/speed-up-your-android-emulator

4.2 Multiplatform Development Tools

Although Java is a powerful modern programming language that is supposed to run on all platforms, it has one major issue. It does not run on all platforms. Neither Windows Phones, nor Apple handheld devices support Java applications (yet alone those written for Android). [29]

Although implementing Eridian just for Android would fulfil the goal of this thesis, the author decided to leave an open door for possible future implementations on other platforms. The second most known option for developing an Android application is coding in C/C++. Being a very old, traditional and almost universal programming language with tons of libraries, one would expect an C++ application to run on all mobile platforms. One would be wrong. The reason is that every platform compiles and runs the application in a different way and that Android, BlackBerry, iPhone and Windows Phone have all different life-cycles. The only way to use one code for all platforms is via a framework. Although there exist some really great game-development frameworks (like Unity, V-Play, Marmelade, Wave, etc.), using such advanced technique together with all the features and possibilities of C++ language might be considered an overkill for 2D board game implementation.

This leads us to look for a higher level programming language. One such language - extremely popular nowadays - is JavaScript [30] [31]. JavaScript is a scripting language used predominantly in web applications. Together with the abilities of HTML5 and CSS3 they form a powerful trio able to run also 2D and 3D games [32].

As a matter of fact, there exists also a very popular mobile development framework based on this web trio, called PhoneGap. PhoneGap creates a mobile application that is run within a web browser (like conventional web applications) but has access to device peripheries, such as camera, GPS, gyroscope, etc. Such an application is able to run on Android, iOS, Windows, BlackBerry, Bada, Symbian and others.

However, since a PhoneGap application is not native and is run via web browser, concerns about its performance arise. When looking at tests, we can see that basic form applications have almost none, or only slight performance issues. These, however, grow with the app complexity and a game written in PhoneGap may have a user experience which is far from optimal [33] [34]. Another issue is an inconsistent user experience. Android, BlackBerry, iOS and Windows Phone all have different user interface, layout and controlling scheme. An application may even be disqualified from app store because of not following the guidelines.

[29] However, Java multiplatform solutions like RoboVM `http://blog.robovm.org/` for iOS and CodenameOne `http://www.codenameone.com/pricing.html` for iOS, Android and Windows Phone are in development

[30] `http://langpop.com/`

[31] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

[32] There exist a lot of 2D and 3D engines for JavaScript, including PlayCanvas, Babylon.js and Three.js

[33] `http://roadfiresoftware.com/2014/04/when-to-use-phonegap-versus-developing-a-native-ios-app/`

[34] `http://www.fastcolabs.com/3030873/our-html5-web-app-flopped-so-we-went-native-and-havent-looked-back`

Moreover, JavaScript is not the best language for large-scale applications, since code management and debugging are much more difficult in a scripting language. This eliminates automatically another player on the field - Appcelerator. Appcelerator is another multi-platforms mobile framework. However, it currently supports only Android and iOS (with BlackBerry and Windows Phone planned). Unlike PhoneGap, Appcelerator creates a native app via a JavaScript proxy, which may have a huge impact on performance in a 2D game (e.g. in image processing). Another problem is that although native UX can be achieved with Appcelerator, it is done by using only those elements which are common to all platforms.

Writing to the lowest common denominator can end up making your application feel foreign to all of them. Applications on Windows Phone are designed to look and behave differently than those on iOS, and that is something that should be embraced rather than glossed over or abstracted away. The experience you present to your users should be the primary concern when designing your application's interface [10].

Thus yet another solution is required. A platform that would allow code reusing across platforms, yet leverage the singularity of each of them. Developed in a language on comparable level as Java, to enable type-safety, good code management and debugging. The language we are looking for is C# on Mono. Exactly that same C# that is supposed to work only on Windows platform. [35]

4.3 Discovering Xamarin

Mono is a free and open source project led by Xamarin (formerly by Novell and originally by Ximian) to create an Ecma standard-compliant, .NET Framework-compatible set of tools including, among others, a C# compiler and a Common Language Runtime. [36].

Mono is an open source, cross-platform implementation of a C# compiler, and a Common Language Runtime (CLR) that is binary compatible with Microsoft .NET. The Mono CLR has been ported to many platforms, including Android, most Linux distributions, BSD, OS X, Windows, Solaris, and even some game consoles such as Wii and Xbox 360. In addition, Mono provides a static compiler that allows apps to be compiled for environments such as iOS and PS3 [9].

The Mono project was started by Miguel de Icaza in 2001, shortly after Microsoft announced the .NET framework. After three years of development as an open source project by Icaza's company Ximian, Mono 1.0 was released. Meanwhile, Ximian was bought by Novell, which was then acquired by Attachmate in 2011. At that time, the

[35] PhoneGap, Appcelerator and Xamarin are not the only representatives of this category. There are several others like Dot42, Rhomobile, Codename One, MoSync... the list is far from exhaustive. However, many of those projects are quite recent or have limited user base. Some have poor performance and quality of their implementation questionable (not all of these negatives apply to all of the multi-platform frameworks). That, and the author's love of C#, made the final leap of faith needed to start coding in Xamarin.

[36] `http://en.wikipedia.org/wiki/Mono_%28software%29`

future of Mono project became questionable. Icaza then announced the foundation of a new company called Xamarin, which later managed to gain perpetual licences for Mono, MonoTouch (Mono for iOS) and Mono for Android [37].

Nowadays, Xamarin offers a whole array of products that help programmers develop mobile applications for iOS, Android and Windows Phone. Let us shortly introduce the products found at their website [38]:

- **Xamarin Platform** - the main platform consisting of C#, Mono, MonoTouch and Mono for Android libraries enabling interoperability with ObjectiveC and Java libraries. It enables development of applications that can be run on Android, iOS and Mac. (Xamarin currently does not support BlackBerry, although it is looking at it in longer term [39]. There exists a project called MonoBerry - Mono for BlackBerry. However, this is not supported by Xamarin [40])
- **Xamarin Test Cloud** - a service for testing applications on real devices via cloud.
- **Xamarin Insights** - a crash reporting solution, tracking also the recent user activity.
- **Xamarin University** - an online mobile application development training.
- **Xamarin Studio** - a brand new IDE for development of Xamarin mobile applications.
- **Xamarin for Visual Studio** - a Visual Studio extension (pre-built in Visual Studio 2013) allowing to develop Android and iOS applications in C#.
- **Xamarin.Forms** - a library of UI elements that is converted to appropriate native GUI (iOS, Windows Phone or Android) upon project build. Xamarin.Forms allow 100% code reuse, since both backend and frontend need to be written just once.
- **Android Player** - Xamarin's own Android emulator
- **.NET Mobility Scanner** - a web application evaluating the compatibility of a compiled .NET code with Android, iOS, Windows Phone and Windows Store
- **Xamarin Profiler** - a tool collecting information about the memory and speed performance of the managed site of an Xamarin application

4.4 The Pros and Cons of Xamarin

Now, let us take a look at what is there to gain and lose when choosing Xamarin instead of native Java. The first things are rather obvious.

[37] http://en.wikipedia.org/wiki/Xamarin#History
[38] http://xamarin.com/
[39] http://www.citeworld.com/article/2114280/development/xamarin-raises-series-b-fund-developer-outreach.html
[40] http://burningsoda.com/software/monoberry/

We gain access to the .NET platform and C# with all its awesome language features like async/await constructs, dynamic typing, LINQ, lambda expressions, true generics, extension functions, delegates and events, properties, and many, many more [10]. At the same time, we do not lose anything from the Android API. The Java classes are either re-written in C# in Xamarin.Android library, or can be invoked directly from C# code. This is the same for ObjectiveC in iOS. Although there are a few known limitations both for Xamarin.Android [41] and Xamari.iOS [42], the great benefit is that instead of learning three languages (Java, ObjectiveC and C#) only one language is enough. The familiarity with the different SDKs is, however, still required.

The other obvious thing to gain is the possibility to target multiple platforms. Note that Xamarin is not a "write once, run everywhere" solution, like PhoneGap. Instead, Xamarin always strives to achieve native look and feel for any of the three platforms. This means, there is different UI and application code for every platform (more on this topic in chapter 4.5) and a shared code for back-end logic [10]. A 100% code sharing may be achieved when using Xamarin.Forms. However, if the application has custom UI elements (like Eridian game has) and platform-specific behaviour, the code sharing may come down to 30-40%.

Now, let us have a closer look at other features and issues that come with Xamarin but may not be so obvious at first glance. Considering speed and memory performance, there is a frightening Debian example [43] showing that C# Mono may be (depending on the program and configuration) 3-5 times slower than the same program in Java. There are rare cases [44] when Mono is actually slightly faster. However, C# always takes less memory (up to 3-times) and there is usually less code needed, as well. Miguel de Icaza (founder of the Mono project and Xamarin) presented his own performance tests in late 2007 [45]. These tests confirm better memory management of C# Mono than Java. The speed tests are mostly comparable (favouring both C# or Java, depending on the test) with two alarming examples, where Java code was 6x and 8x faster.

Both memory consumption and speed are crucial factor on mobile platforms. So how does Mono fare there, especially in Android? Unfortunately, there do not exist any official benchmark tests. Neither is it in the scope of this thesis to perform reliable formal tests comparing performance of native Java code to Xamarin Android code.

As there was no Java implementation to make any comparison, the following information are based just on assumptions and subjective impression. The Eridian board game implemented with Xamarin.Android has look, feel and performance as if developed natively. There is a small delay at start up, ranging from 0,5-2 seconds. This issue was solved with a splash screen, as advised by Jonathan Pryor. It is true that the application takes up 27MB when installed. However, 25MB from that amount is consumed by the in-game images (sprites, maps, etc). Once the application is started, it runs very smoothly and there are no performance issues, neither in speed, nor in memory consumption. The

[41] http://developer.xamarin.com/guides/android/advanced_topics/limitations/
[42] http://developer.xamarin.com/guides/ios/advanced_topics/limitations/
[43] http://benchmarksgame.alioth.debian.org/u64q/csharp.html
[44] http://benchmarksgame.alioth.debian.org/u32q/csharp.php
[45] http://tirania.org/blog/archive/2007/Dec-28.html

most comprehensive tests were made by the user gregko on StackOverflow community [46]. The tests consisted of string and file processing, done 5 times for Java and C# on several real devices, as well as emulators. The tests were repeated after some period. Devices with older versions of Android running on ARM processors showed that Mono was significantly slower than native Java. However, Intel processors displayed comparable performance. And with newer versions of Android (4.0 and above), C# Mono was always faster than native Java code. The overall average of all tests show that a C# application runs only 93% of the time consumed by Java application. There exist several other tests that come with the same results - Xamarin Android is comparable to Java. It is often slightly faster, however, there are few exceptions where it is significantly slower [47][48].

Back in 2011, Android applications written in Mono C# used to have around 3 seconds start up penalty [49] as pointed by a member of Xamarin team, Jonathan Pryor. Since then, this time has been significantly reduced. Nonetheless, large applications might still have a lag upon starting. Using a splashscreen is advised in such cases, lowering the psychological impact. There exist some complaints about increased size of Mono applications [50]. Although there definitely is a slight overhead (the Mono runtime is always packed as a native library [9]), the final size of the application is largely dependant on the settings used for a Linker during compilation [51].

There are other issues connected with Xamarin as well. As every software product, Xamarin is not bug-free [52], although the bugs caused by Mono for Android are rare and occur mostly in uncommon scenarios. During the implementation of Eridian, the author encountered only a pair of Mono-related bugs. They were easily overcome with a simple workaround. There are a few small inconsistencies in the Xamarin Android base library implementation (most notably, some accessors are re-written as C# properties, while others retain their Java get/set method form - this is affected by Xamarin's Java to C# transformation rules [9]).

Although the Xamarin community is considerably smaller than Java Android community, it is large enough to provide support in times of dire need. Xamarin provides a component database and actively encourages developers to write new components, often motivating them with prizes for the most interesting submissions. Xamarin also features a support community. Business and enterprise developers have direct access to Xamarin support team, while the rest of developers take care of each other. More often than not, when looking for a programming solution for Xamarin Android, there are only examples from Java Android available. This is not a huge problem since C# and Java are syntacti-

[46] http://stackoverflow.com/questions/17134522/does-anyone-have-benchmarks-code-results-comparing-performance-of-android-ap

[47] https://github.com/EgorBo/Xamarin.Android-vs-Java

[48] http://sysmagazine.com/posts/194866/

[49] http://stackoverflow.com/questions/7538607/is-monodroid-slower

[50] http://www.whitneyland.com/2013/05/why-i-dont-recommend-xamarin-for-mobile-development.html

[51] http://developer.xamarin.com/guides/android/advanced_topics/linking/

[52] https://bugzilla.xamarin.com/

cally similar, so it is easy to rewrite a Java solution to C# code. However, it might be a bigger issue in iOS development, since ObjectiveC is significantly different.

Another great concern is that there might be a lag between the official release of a new Android (or iOS) SDK and Xamarin's software update. This concern is unsubstantiated. Generally, Xamarin releases product updated within 24 hours after their official release by Google or Apple [10], since their team tracks the beta releases.

4.5 Mulitplatform Implementation Specifics

As mentioned earlier, Xamarin does not strive to achieve "code once, run everywhere". It is true that this can be achieved with the Xamarin.Forms library. Yet, they were developed much later than the original MonoTouch (Xamarin.iOS) and MonoDroid (Xamarin.Android) libraries. Moreover, if the application uses more than just the lowest common denominator forms of all platforms, Xamarin.Forms will be not enough. Specifically, in our case, game menu, login and victory screens would all be feasible with Xamarin.Forms. However, the main game view with our 2D implementation of Eridian board game required the creation of a custom view.

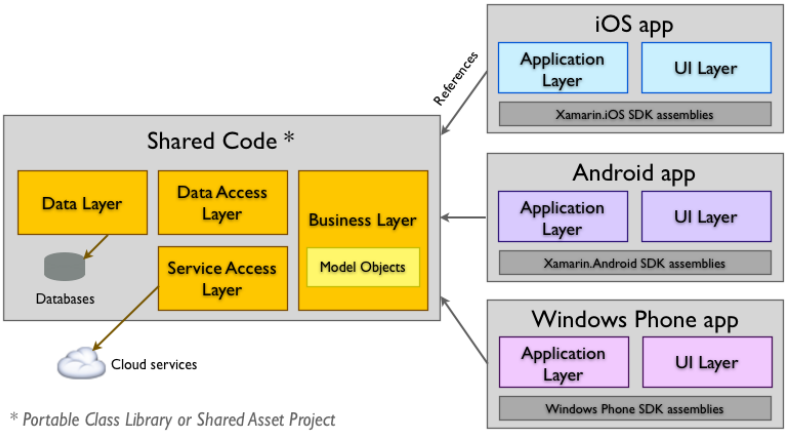

Figure 2: A Xamarin multiplatform application architecture (Xamarin.com)

Yet, to leverage the possibility of sharing as much code as possible, Xamarin has developed a series of recommendations for software architecture for multi-platform mobile applications [53]. This architecture requires dividing the multi-platform application code into several parts. The central core project (or assembly) contains all the platform-agnostic classes and methods. Except for the main Business layer code, it may contain an SQLite database, or a different data storage file (XML, Json) and corresponding Data Ac-

[53] http://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/sharing_code_options/

cess Layer. When developing an application communicating via the Internet, the client code or service layer should be included within the core assembly as well [54].

This core project (or assembly) is then referenced by 3 platform specific projects - one for Android, one for iOS and one for Windows Phone. Those three assemblies contain the user interfaces, native for each platform. They may also contain event handlers, resource management, platform-specific APIs, life-cycle event methods and others.

The core project can be either a Shared project or Portable class library. The Shared project is a simplistic approach in which the whole solution may, in fact, be composed of a single C# project. This is possible thanks to the use of compiler directives, such as #if __ANDROID__, telling the compiler which portion of the code should be compiled for which platform.

Portable class library (PCL) approach [55] requires a better architectural division of the code. PCLs allows to write a platform-agnostic code which can be later used by any platform. The downside is that not all classes and methods of the .NET platform are available.

[54] Note the terms project and assembly are interchangeable in this context. It is because a solution may consist of both 4 projects or 4 assemblies, each containing several projects

[55] `http://developer.xamarin.com/guides/cross-platform/application_fundamentals/pcl/introduction_to_portable_class_libraries/`

5 Server side: PUSH Communication Model

To fully understand the following chapters and decisions made, some implementation details need to be clarified (further details on implementation will be given in the next part of the thesis 6). In the first three parts of this thesis we have decided to create Eridian - a multiplayer cooperative board game implementation for Android platform. We have also chosen to use C# on Mono for Android rather than classic Java for Android.

As the game will be played by multiple players and on mobile devices, a sort of inter-device communication needs to be established. A client-server architecture was favoured over peer-to-peer architecture. This is because Wi-Fi connection is more stable than Bluetooth, although it is not as available. Yet, the Internet connections are also faster and allow players to be far away from each other.

In a client-server model, we need to decide, what responsibilities will be given to the client and what to the server. In other words, will we have a thick, thin or hybrid client? There are certain benefits for both thick and thin client, but it was decided to take the golden mean and implement a hybrid client.

To clarify things a little bit - if Eridian client was thin, its only responsibilities would be to send user input to the server and display the response message. However, since our client is hybrid, it is capable of doing more. Eridian client also stores a local copy of the game played. Thus, during an update, only the changing data are being transferred, which lowers the traffic. Only in case of some disturbance (such as temporary connection failure) the server sends a complete copy of the current game state.

Thanks to the local game copy, the client is also able to check some non-concurrent conditions. For example, if a player touches the screen, the client is able to check, whether given point is a possible movement location. However, checking whether some action does not infringe the rights of another player is a responsibility of the server.

To store the game state, a sort of persistence is required. Database is one option, but the speed of concurrent reads, writes, updates and deletes is quite performance-costly. Another option is to store them only as in-memory objects. However, concurrent environment provides the opportunity for unexpected behaviour, due to dirty writing, phantom objects and value/reference errors. And during a transport between the client and the server, they need to be serialized anyway. Thus the decision was made to store the current game state as an XML structure. Since C# was chosen as the primary development language, we can leverage the functionality of LINQ to XML, which makes all our work much easier.

5.1 TCP and UDP

There are two basic protocols that enable communication between various devices via the Internet. TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). In short, it can be said that UDP is faster and less reliable (because it does not wait for confirmation replies) while TCP is slower but more reliable (because of checking the confirmations) [3].

In case of fast paced real-time multiplayer games with a lot of concurrent user input, there is a discussion whether it is better to use UDP or TCP. However, in the case of a slow-paced turn-based multiplayer game, the choice is quite obvious. The traffic is far from heavy, small delay is acceptable but data loss can corrupt the gameplay. So, it is TCP for Eridian.

The implementation of a custom TCP communication - when done properly - is rather complex and time-consuming. A simple socket-opening and data-sending is just the beginning. Events like drop in connection speed, unstable connection, disconnection and reconnection have to be dealt with. Custom communication protocols have to be implemented, dealing with the problems of possible packet loss. The server has to keep track of connected users (players), which of them got a message and to what group (game room) they belong - so that a move done by one player can be seen by all other players in the same game.

Implementing all the functionalities that are needed to cope with the unpredictabilities of the Internet is out of the question. Instead, let us look at solutions that would help us skip the hard work process and jump right where the fun begins - sending and receiving messages.

5.2 Web Services

Since we have transformed the current game state into an XML structure, which will be sent between the server and the clients (either as a complex object, or just the changing parts), there is one technology that instantly comes to mind. Web services.

Web services are application components which communicate using open protocols. They can be used (consumed) by other applications. They are based on HTTP and XML (or JSON). As the client side is implemented in Mono for Android C#, there are three types of web services that are available for us [56].

RESTful services (or simply REST) offer a platform agnostic, stateless, cache-able and simple to use client-server architecture over the HTTP protocol. The client sends a request to a parametrized URL address. Based on the individual parameters, the server processes this request and sends appropriate answer (in form of an XML or JSON message).

SOAP is another type of web service. Instead of parametrizing a GET request, it relies on sending a POST request with an XML/JSON object. This object is constructed according to a pattern described on the server, written in WSDL (Web Service Description Language). Since this pattern is exposed by the server, a proxy object can be generated by the IDE, making the coding process easier.

The last option available - in our case - are **WCF services**. WCF services is a Microsoft specific technology, exposed in Xamarin via the BasicHttpBinding protocol. In fact, they

[56] `http://developer.xamarin.com/guides/cross-platform/application_fundamentals/web_services/`

can be both REST and SOAP services that are covered by a common interface. The use of WCF also involves creating a proxy and generating appropriate objects used for the communication.

5.3 Duplex Web Services and Mono

Each of the previously described web service technologies has its positive and negative sides. However, they all share a common issue that is crucial to our implementation. All of them, REST, SOAP and WCF are HTTP-based. This means, they all follow the PULL model in which the client is the only active part and the server simply responds to its messages. In other words, the server is not able to send updates to players, if a request was not made. In practice, this would look like the following example:

Player A makes a move, sending a request to the server. The server responds, thus player A can see his/her progress. However, since players B and C were not on their turn, they could not make a request and they will not see the result of player A's action. Only when they are on turn, they can make move, send request and the server will respond showing the result of their action and also, the result of player A's action that he/she has made several turns before. Similarly, the user A will not see the moves of players B and C, until it is his/her turn again. This is like playing a game blindfolded.

There certainly are ways to get passed this unwanted behaviour. One solution that sounds very simple and effective, is periodically sending requests to the server for possible updates. This method is called polling [1]. In practice, a background timer thread would be created, sending requests to the game server and then parsing the response.

Although polling seems good enough, it does not come for free.

Constant connections and disconnections have a high cost in terms of bandwidth and processing at both ends of communication. The worst part is that this cost increases proportionally to our need for faster updates and the number of clients making use of the service at a given time. [1]

The first idea that pops up to solve this problem is lowering the number of requests by making the intervals longer. However, there would still be a lot of requests. Moreover, the user experience would be affected negatively. In fact, as long as we stick to the PULL model, there is no ideal way out. The only solution is to use the PUSH model. A communication model in which the server can send updates at its will as well as the client's.

So, we are looking for a two-way web service. As a matter of fact, Microsoft has developed a duplex communication model for the WCF services [5]. This implementation allows an immediate server-to-client communication without previously made requests. This duplex web service works seamlessly in standard Windows environment. However, under the Mono runtime, there is a little catch.

Microsoft duplex WCF services use the WSDualHttpBinding class. And this class is precisely one of the few elements that are neither implemented in the Mono runtime, nor

are planned to ever be [57]. Moreover, even if the duplex web service communication was available, there would still be some remaining issues, like connection management, user and group management, request processing and immediate result broadcasting.

We have found ourselves in similar situation, as during the client side development in chapter 4.2. There exists a conventional solution with some limitations (Java for Android does not work on other mobile operating systems / custom TCP works well, but takes a lot of time, resources and dedication to implement). We have discussed one or several alternative solutions that eliminate some issues, but create a lot of others (PhoneGap works on more platforms, but has bad performance / WCF cannot be duplex on Mono). And now we are standing before the best mean how to solve our problems. It was Xamarin's Mono for Android on the client side. And it is ASP's brand new framework, SignalR that will help us on the server-side.

5.4 Discovering SignalR

SignalR is a framework that facilitates building interactive, multiuser, and real-time web applications (although not only web applications), making extensive use of asynchrony techniques to achieve immediacy and maximum performance [1].

SignalR is a server-side library belonging to the ASP.NET family. It was released only quite recently, in January 2013. SignalR enables developing both client and server-side solutions for a HTTP communication based on the PUSH model. It covers several connection types and is able to automatically switch between them. During the initial negotiation between the client and the server, SignalR detects what connection types are supported by both ends and chooses the most efficient [1]. At first, it tries to establish a connection based on Web Sockets, since this is the only technology that supports full-duplex on the same channel and is therefore the most efficient. In case they are not available, the connection falls back to Server-Sent Events, forever frame or long polling.

This is just the beginning. SignalR has a lot more to offer. The framework tracks user connections giving them unique IDs. There is also the possibility of custom implementation of IUserIdProvider interface, enabling to target the users by their user-names, logins, emails etc. These users can further join and leave one or several groups. Because the group membership is based on the Publish/Subscribe pattern [58], it enables immense scalability. A small drawback of the Pub/Sub model is that we are unable to keep track of the members of individual groups. This issue is easily bridged by implementing a ConcurrentDictionary as we will see in the implementation part, in chapter 6.4.2. [1]

SignalR is also able to handle various transport-related events, like connection, disconnection, reconnection, drop in connection speed, errors, state changes, etc. We can also access two levels of abstraction on the server side. The PersistentConnection class is a lower-level API that offers similar coding abstraction like when working with sockets

[57] `http://www.mono-project.com/docs/web/wcf/#components-with-no-plan-to-support`

[58] `https://msdn.microsoft.com/en-us/library/ff649664.aspx`

[1]. On the other hand, Hubs (the other server class) provide a much higher-tier abstraction above the Internet protocols and transports.

Hubs use an imperative development model based on the flexibility of JavaScript and the dynamic features of C#, which creates an illusion of continuity between the client and the server, two physically separated environments [1].

When creating and calling server methods from the client (and vice versa), SignalR leverages the possibilities of the `dynamic` keyword introduced in .NET framework 4.0 [12]. Dynamic objects are handled as if they were a part of a scripting language, thus they are not subjected to type control and it is assumed they support any operation at compile time. When a server method is called from the client, it is transformed into a HTTP request. At server side, SignalR tries to locate a method with the same name and same number of parameters (note that only parameter number is important, not their type). If such method exists, it is immediately invoked. The whole process works the same way when calling client methods from the server [1].

6 Implementation

The following chapters will focus on details of implementation. No introductory part for C#, .NET platform or Android development is provided. Yet, solid knowledge of C# and (at least) basic knowledge of Android development is required for the reader to fully understand the content of this part of the thesis. A great introduction into Android for C# developers can be found in the book Xamarin Mobile Application Development for Android[9].

6.1 Class Model

To begin describing the client, server and communication implementation, we need to familiarize ourselves with the core model of a game at first. This model stores the current game state information and can be serialized into an XML structure. The resulting XML (or its parts) is then sent between the server and the client.[59]

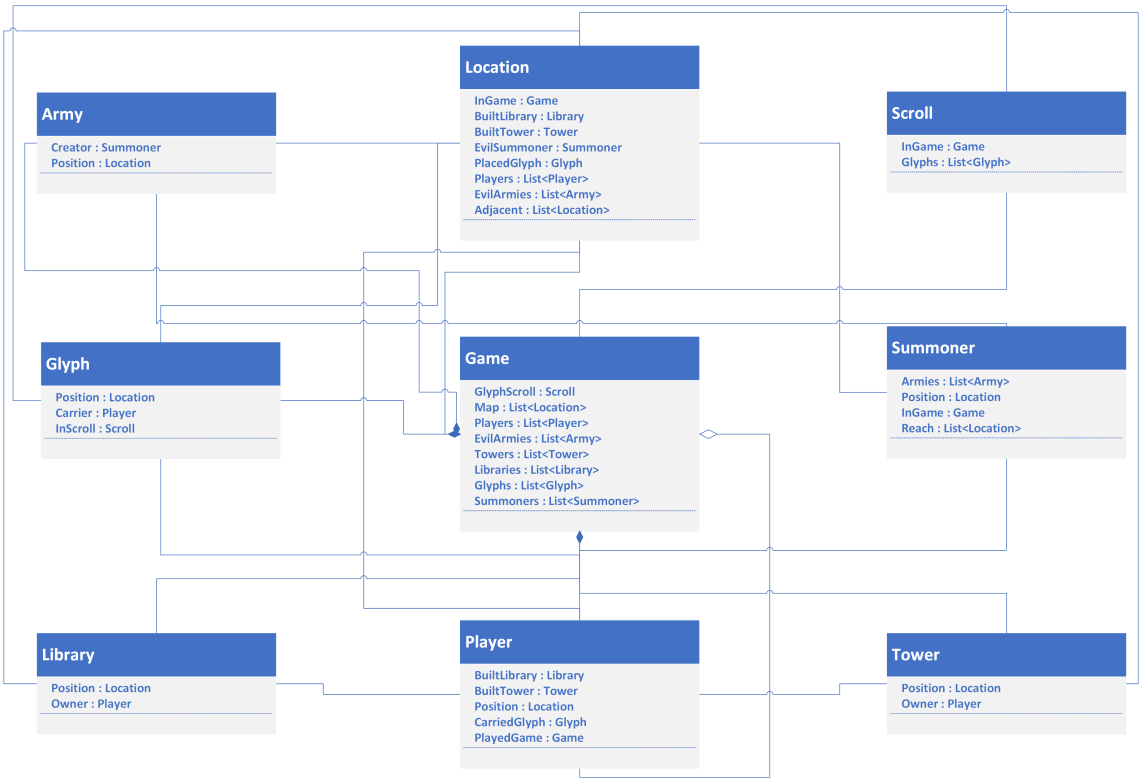

Figure 3: A simplified diagram of Eridian class model

[59]Note that the list of properties in this diagram is not exhaustive. Only properties referencing other classes from the class diagram are depicted.

6.1.1 Game Class

The core class of a game is the Game class. The Game class follows the Data Transfer Object (DTO) design pattern. It stores all objects, to which it has both direct and indirect relationship, in a form of .NET Lists. The reason for implementing a Data Transfer Object is that the classes within Eridian game model have very complex relationships. There are several objects that have (or should have) reference to a common component. In this place it is best to illustrate an example:

The Player class and the Location class both have reference to a BuiltTower (instance of the Tower class). This BuiltTower has also references to its Owner (the Player) and position on which it is built (the Location). In the case a new Tower object is created, we would have to add new reference to the Player, Location and then reference them both inside the Tower. The same would have to happen in case of a Location or Player update. As if it was not complicated enough, the server is an environment with high concurrency, so unpleasant events like Phantom objects or Dirty writing could occur.

With the combination of Data Transfer Object and Identity Mapping design patterns and the power of LINQ and C# properties, this complex problem can be solved very elegantly and easily. When a new Tower object is created, it is inserted only into the List of Towers inside the current Game instance. The mapping of this Tower object from a Location instance is then simply performed as a result of query comparing the current Location's ID and the ID of the position on which the Tower was built. Analogically, the Tower built by a Player is simply an instance from the Game's list of towers, which has the same Owner name as is the name of the particular Player.

```
public Tower BuiltTower
{
    get
    {
        if (PlayedGame == null) return null;
        return PlayedGame.Towers.FirstOrDefault(tower => tower.PositionID == this.ID);
    }
}
```

Code sample 1: The getter property for a BuiltTower from the Location class

PlayedGame is another Location property, which returns the instance of the Game that is played (in other words, the Game instance that holds the particular Location in a list). Towers is the List of Tower objects inside the current Game. And thanks to LINQ we can use a simple lambda function comparing the "foreign key" PositionIDs of individual Tower objects with the ID of the current Location instance. Moreover, since the Game class stores all the Locations and Players, we can implement reverse accessors - the Owner and Position getters inside the Tower class - as well.

This approach ensures that there will always be the same Tower object for particular Player on particular Location. The reader may have noticed the PersistentContext and the LoggedPlayer properties. PersistentContext is - both in the server and client implementation - a static class which holds the current game state. The LoggedPlayer is a

client-specific instance of the Account class. Both implementations of PersistentContext and the Account class will be discussed in following chapters.

```
public Location Position
{
    get
    {
        return PersistentContext.LoggedPlayer.PlayedGame.Map.FirstOrDefault(location => location.
            ID == this.PositionID);
    }
}
```

Code sample 2: The getter property for the Location of a Tower object

The other properties of the Game class include an ID which uses the C# Guid creator, name of the game, player limit and current in-game day (turn). Each Game instance also contains its own data serialized in an XML string.

6.1.2 PersistentContext - Server-side

The core functionality of Eridian's server is provided by the GameHub class, which inherits from the SignalR Hub class. The Hub class works in a similar fashion as ASP WebForms pages. On every new client request a new instance of the Hub is created. This instance processes the request, sends a response and then is destroyed. When the client sends another request, this new request is processed by a brand new instance of the Hub class, which has no knowledge of the previous Hub objects.

Just to illustrate, what the previous paragraph means in practice, let us examine an example. Players A and B are playing a game of Eridian. Player A's device connects to the server and the transport technologies are agreed upon in the initial handshake. A new instance of the GameHub class is created to handle this new connection. However, the GameHub is dropped immediately after it performed the method. When player B connects to the server, the same process repeats from the beginning - just with a brand new instance of the GameHub class. This new instance has no knowledge of the previous instance, servicing player A.

Now, player A is on his/her turn once again. He performs a move, the client application sends a request to the server and another instance of the GameHub class is made - third one in a row. And, just like in the previous case, this new GameHub has no information about any states, data or even the existence of the previous GameHubs.This means no state is kept on the server between the individual request processes. The only way how to retain information and share them among the respective GameHub instances is by using static objects. [1]

Another feature of the SignalR framework is the Pub/Sub model. This allows clients to be grouped together and then broadcast data to whole groups rather than to individual connections. Yet, the group membership information is carried only in the requests and responses and is not retained on the server. To keep track of which connection belongs to which group, custom lists of connections and groups would have to be created.

As a matter of fact, the SignalR specification discusses this topic and it advices to use a static ConcurrentDictionary [60]. And that is where the server-side PersistentContext class comes to importance. The server side PersistentContext class is a static class containing two static ConcurrentDictionaries - one for players (or SignalR connections) and the other for game rooms (or SignalR groups). The PersistentContext is called from various methods of a GameHub and since it is static, all the information from the various GameHub instances are retained together in one place.

Let's add the PersistentContext to our example. Player's A and B may all be serviced by different GameHub instances, but they still are represented by Player objects within the static PersistentContext ConcurrentDictionary. And they both participate in the same instance of a Game, which is located in the second static ConcurrentDictionary of the PersistentContext.

6.1.3 PersistentContext - Client-side and the Account Class

The client-side PersistentContext serves to a similar purpose as its server-side counterpart. It stores player and game information throughout the various Android activities. However, it does not contain two static ConcurrentDictionaries, but only a single static Account class.

The Account is a client-specific class that is absent on the server side. This is due to the fact that the server needs to hold several instances of games and players that are playing, or waiting in the lobby. However, the client is able to play just one game at a time. Therefore, the client PersistentContext needs to keep track of only a single game instance, and the information about the current device user. They are all stored as properties of the Account class:

- **Name** - the login name of the user of the device
- **PlayedGame** - the instance of the game in which the user participates (or NULL, if he/she does not play any game at the time)
- **MyCharacter** - a read-only property, which returns the Player object from PlayedGame .Players list which has the same name as is the Account name

6.1.4 Player Class and IUserIdProvider

The Player class represents the user and his character in the game. Unlike the other classes, the Player class does not use a Guid as unique identifier, but rather a string Name (which is the same as the Account.Name of the particular user). The reason for this is that Eridian's custom implementation of IUserIdProvider interface, which is used by SignalR as custom user recognition, uses the player login to address him/her.

By default, SignalR distinguishes users by their ConnectionID. The ConnectionID is a Guid-like string that is generated during the negotiation phase of connection establishment between the client and the server. However, SignalR is unable to track the physical

[60] https://msdn.microsoft.com/en-us/library/dd287191%28v=vs.110%29.aspx

identity of the client's device, therefore each new request creates a new ConnectionID. In a web application, all asynchronous requests would have the same ConnectionID until the user navigated onto a new web page. At that point, the old connection would be closed, new connection would be opened and the user would get a new ConnectionID.

In an Android client, there is theoretically just a single connection from the application start till the application end. There can be some disturbances in the connection, but SignalR is able to reconnect after short period of disconnection. However, Android applications can be stopped or paused because of various other reasons, like incoming phone call, switching to another application etc. When an Android application is resumed, the whole view is recreated. This creates an interesting situation for the SignalR technology.

Since the application was not destroyed, the server is still trying to reconnect to the old ConnectionID. However, as the View was recreated, a new ConnectionID was established as well, co-existing with the old one. The client is trying to reconnect to the game, but is unable to do so, because his/her ConnectionID is different from the old ConnectionID. This deadlock situation lasts for several minutes, until both reconnection attempts run out of time. After that, the client is able to connect with a brand new ConnectionID (third in a row). Yet, this last ConnectionID is again different. The server will treat it as a new client and will not let the player finish the game he/she had started.

To prevent this unwanted behaviour, SignalR provides the possibility to address users via a custom implementation of the IUserIdProvider. This is done by creating a custom class that inherits from the IUserIdProvider and implements the GetUserId method:

```
public class GameUserProvider : IUserIdProvider
{
    public string GetUserId(IRequest request)
    {
        if (request == null)
        throw new ArgumentNullException("request");
        else if (request.Headers != null && request.Headers["userName"] != null)
        return request.Headers["userName"].ToString();
        else return null;
    }
}
```

Code sample 3: Eridian IUserProviderImplementation

In our case, we will use the player login as the unique identifier. This way, a player with the particular username will be able to reconnect and play the game, regardless of his ConnectionID. The username is sent via the custom HTTP header:

```
_connection.Headers.Add("userName", name);
```

Here, _connection is the instance of HubConnection SignalR client class and name is the login passed as a parameter.

6.2 Splashscreen

The game model is now clear. We can continue in the description of the game implementation. However, we will not describe individual assemblies and classes separately.

Although this approach is more conventional, it would be harder for the user to understand all the interactions. Therefore, we will follow the Android activities, screen by screen and describe individual processes that are involved.

Since all Xamarin.Android application need to initialize the Mono runtime at their start up, there might be a small lag at the application start. To lower the psychological impact and smooth the user experience, a splash screen is advised to be incorporated.

A splash screen is an empty screen that usually displays only the application logo. However, this logo should not be a Bitmap or an ImageView. Instead, the splash screen should use a custom theme. The reason for that is, Android is able to display the theme immediately, even before loading the application. The Eridian splash screen theme is rather a simple one:

```
<style name="Theme.Splash" parent="android:Theme">
<item name="android:windowBackground">@drawable/splash_centered</item>
<item name="android:windowNoTitle">true</item>
</style>
```

Code sample 4: Custom theme for the splash screen

No layout is required either. We simply create a splash Activity, use the custom theme and then start the next activity:

```
[ Activity (Label = "Eridian", Theme = "@style/Theme.Splash", MainLauncher = true, Icon = "
    @drawable/iconsmall", NoHistory = true)]
public class SplashActivity : Activity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        Thread.Sleep(250);
        StartActivity (typeof(LoginActivity));
    }
}
```

Code sample 5: The Activity code for the splash screen

If the reader has developed an Android application in Java before, he most certainly has observed a few differences. Besides the syntactic ones (**override** keyword instead of @override annotation) and conventions (upper and lower camel case), the most notable difference is the [Activity] attribute right above the class declaration.

Instead of creating a global Application Manifest, in Xamarin.Android we can define the Activity preferences right above the class declaration [61]. It can be argued that this approach is better, since all the code involved is seen together.

6.3 Login Activity

The SplashActivity had no functionality at all, nor involved any server methods. It only served to load the LoginActivity. This one is a bit more complicated. The LoginActivity

[61] It is worth to note that during compilation the traditional Application Manifest is still generated by Xamarin, bringing together individual class attributes.

features a single spinner (Android equivalent to a dropdownlist or a combobox) and a single button. In the OnCreate method, the spinner is filled with all emails that are used as an account in the particular Android device. To retrieve the emails, a simple lambda function is used:

```
Android.Accounts.AccountManager
    .Get(this)
    .GetAccounts()
    .Where(item => item.Name.Contains("@"))
    .Select(item => item.Name)
    .ToList();
```

Code sample 6: A combination of LINQ and lambda functions

When the user chooses the email he wants to use as his in-game name, he clicks on the login button. At that point an asynchronous event is fired, using the C# async/await construction. In the event, the network availability is checked and right after, a connection with the server begins to establish. A new instance of GameClient class is created. In the constructor, the GameClient class creates a new SignalR HubConnection and a proxy derived from it. Also, all the GameClient events are wired up to the calls made from the server. We will illustrate this on a code sample:

```
public class GameClient
{
    private readonly HubConnection _connection;
    private readonly IHubProxy _proxy;
    public event Action<string, string, int> OnTokenChange;

    public GameClient()
    {
        _connection = new HubConnection("http://192.168.1.102:80");
        _proxy = _connection.CreateHubProxy("GameHub");

        // the following piece of code corresponds to the server calling a SendTokenChange(string,
            string, int) method
        _proxy.On("SendTokenChange", (string oldPlayer, string newPlayer, int actions) =>
        {
            if (OnTokenChange != null)
                OnTokenChange(oldPlayer, newPlayer, actions);
        });
    }
}
```

Code sample 7: Initialization of methods and events in the GameClient class

If the server sent a message calling the SendTokenChange method, the OnTokenChange event will be fired immediately, notifying all subscribers about the token change. But let us go back to the login process. Right after the GameClient instance is established, its Login method is called.

```
public async Task<bool> Login(int waitMilis, int version, string name)
{
    var cts = new CancellationTokenSource();

    try
    {
        cts.CancelAfter(waitMilis);
        if (!_connection.Headers.ContainsKey("userName"))
        {
            _connection.Headers.Add("userName", name);
        }
        await _connection.Start();
        return true;
    }
    catch(Exception ex)
    {
        CallFailure(ex);
        return false;
    }
}
```

Code sample 8: Asynchronous Login method

The Login method again implements the async/await construction, returning a Task object of the bool type. Simply said, if the connection is started within given time limit, it returns true, else false is returned. If the connection start was successful, a new Account class is created within the client and the LobbyActivity is created. So far, no custom code was necessary for the server side, since everything was handled by the SignalR Hub class and our custom IUserIdProvider.

6.4 Lobby Activity

The LobbyActivity is again a more complicated Android activity. It consists of three logical sections - the tutorial section, available games section and new game section.

6.4.1 Lobby Activity - Tutorial Section

The tutorial section consists of two simple buttons. One button starts the HowToPlayActivity and the other the TutorialActivity. We will not further discuss those two Activities, since HowToPlayActivity is a simple ScrollView with written game instructions and the TutorialActivity is a simplified version of the GameTableActivity, which will be described later on.

6.4.2 Lobby Activity - Available Games Section

The available games section consists of a ListView and a Button for refresh. When the player enters the LobbyActivity, or clicks on the refresh button, the asynchronous GetAvailableGames

() method is called. This method is never awaited (using the await keyword), so the user interface is not blocked when a request for new games is sent.

The LobbyActivity.GetAvailableGames() method checks the Internet connection, registers to the client's OnGamesAvailableReceived event and then calls the GameClient. GetAvailableGames (**int** waitMilis) method. This method sends request to the server, calling a server method with the same name.

When GameHub.GetAvailableGames() method is called on the server, the server selects all the games in its PersistentContext, which have not reached the player limit. After that the server responds to the calling client by calling the SendAvailableGames(IEnumerable< Tuple<**string**, **string**>> gameList) method. This method was wired up in the client with the OnGamesAvailableReceived event in the GameClient constructor. And since the LobbyActivity .GetAvailableGames() subscribed to this event, it will get the list of available games, which fills the adapter of the ListView.

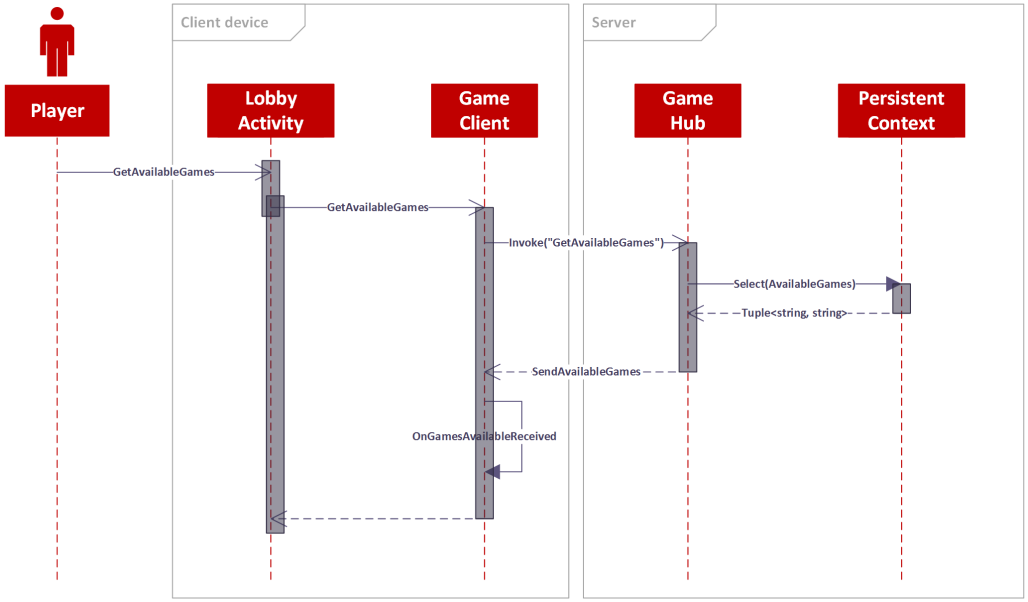

Figure 4: Available games communication sequence diagram

6.4.3 Lobby activity - New Game Section

The last section of the LobbyActivity is dedicated for starting new games. It contains a TextEdit for the game name and 3 Buttons - for a 2-player, 3-player and 5-player game variation. When any of those buttons is clicked, the client checks whether there is a name in the TextEdit, checks the connection and then calls the GameClient.StartNewGame(**int** waitMilis, **string** player, **string** gameName, **int** maxPlayers) method.

The rest of the communication follows similarly as when pressing the refresh button that gets available games. The client calls the GameHub.StartNewGame(**string** playerName, **string** gameName, **int** maxPlayers) method on the server. The server checks, if a game with

the same name does not already exist. A new instance of the Game class is created, it is added to the ConcurrentDictionary holding all games and the player is assigned into the game.

The game instance is then serialized and sent to the client via the SendGameJoined(**string** xDoc) method. Back on the client side, this method fires the OnGameJoined event, to which the LobbyActivity subscribed a method. If no error message was detected by the client, the received XML is parsed into a Game instance, which is then assigned as the PlayedGame in the Account class. Eventually, the CharacterActivity is started.

6.5 Character Activity

The purpose of the CharacterActivity is to allow the player choose his game character. This is achieved with a Spinner that is filled with the names of available characters. On selecting an items, the character's description and in-game appearance is displayed. The player is also able to see the names, colours and characters chose by his co-players.

When the player is happy with his choice, he clicks the start button. This sends a request to the server via the GameClient.ConfirmCharacter(**string** player, Guid gameID, Enumerations .RaceType race, **int** waitMilis) method. The server checks, if all players have confirmed their characters, and if so, the PlayerUpdate(**string** xDoc, **bool** start) method is called back on the client. This fires OnPlayerUpdated event, upon which the CharacterActivity starts the GameTableActivity.

6.6 Game Table Activity

Finally, we have arrived at the point where the main game View can be presented. This View is called GameTableView, and it will be thoroughly discussed in the following subchapters.

6.6.1 Choosing WebView

When developing a 2D game for Android, the need for a specialized View will sooner or later become a crucial factor. Although the custom class can derive directly from View, it can be helpful to choose a View descendant that already implements the desired functionality.

For the Eridian game, we will need a View that is able to display a huge image (the game map). The map should be zoomable and scrollable. Players should also be able to move their characters simply by clicking on the desired locations. Author's first intent was to implement a simple ImageView. However, Android's ImageView can neither zoom, nor scroll. There are two ways how to solve this problem. The ImageView can be either expanded with the zooming and scrolling functionality, or another View with that functionality will be used.

For Java Android, there already exist several implementations of scrollable and zoomable ImageView. However, for Xamarin.Android the range of third-party libraries is a bit lim-

ited. The first implementation that was tested, was ImageViewExt [62], by Takeshi Hiruta. ImageViewExt is a simple extension of ImageView that supports scrolling, but not zooming. It is also possible to get past the borders of the image and the process of scrolling was lagged on the tested device (Sony Xperia X5303).

Another possibility for Xamarin was the port of Java's TouchImageView [63]. This component already implements all the desired functionality (both zooming and scrolling). However, the response time was still very high and scrolling was again very laggy.

Another alternative for a scrollable ImageView is to use a WebView. Although the default purpose of a WebView is to display Internet content in a web browser, it can be also used in different applications. WebView implements advanced functionality of both zooming and scrolling. And, when compared to previous custom Views, it is very smooth and fluent, with immediate response time.

Of course, so far we have been duly ignoring almost every Android game-development guidebook [4][13][11]. When reading any Android 2D game tutorial, you will always be told to use SurfaceView. So, why will Eridian not simply follow the same approach? Let us examine, what does SurfaceView offer compared to WebView.

For the majority of Android Views (WebView included), the only mechanism for redrawing is by calling the Invalidate method from the UI thread. Unlike the majority, SurfaceView offers the functionality to redraw in the background, thus the UI thread is not blocked. Thanks to SurfaceView, all the heavy rendering can be done independent on the main UI thread.

However, since SurfaceView has its own dedicated surface buffer, its resource consumption is higher than that of a normal View. SurfaceView cannot be hardware accelerated (WebView can), and although it can be redrawn anytime, the holding View itself will redraw only after the VSYNC event has arrived. Compared to WebView, it is also more difficult to implement a SurfaceView. In WebView, only the OnDraw method has to be overridden. SurfaceView requires the implementation of SurfaceCreated and SurfaceDestroyed methods, creating a render thread and synchronizing it with the main thread [64] [65]. Thus, adding a SurfaceView in a game is efficient only in the case of permanent heavy rendering. And since Eridian is a turn-based game, we have to fear no such thing.

6.6.2 The OnDraw Method and Image Handling

The main View that is used in the GameTableActivity is - in fact - a composition of two Views, held in a common containing View. We will analyse this layout in the next chapter, but now let us focus on one of the components, the GameView.The GameView derives from the WebView (as we have learned in the previous chapter). We will focus specifically on the overridden OnDraw() method and all other methods that are called from it.

[62] http://stackoverflow.com/a/13218033/1942656

[63] https://gist.github.com/justintoth/10663869

[64] http://source.android.com/devices/graphics/architecture.html

[65] http://pierrchen.blogspot.cz/2014/03/anroid-graphics-surfaceview-all-you.html

In the constructor of the GameView class, 5 Bitmap Dictionaries are initialized. These Dictionaries serve to store the spritesheets for player's characters, towers, libraries, enemy summoners and armies. And since WebView can be hardware accelerated, there is a lot of resources saved on the zooming and scrolling of the large map image, since its RenderLayer can be cached [66].

On each call made to the OnDraw method, the method body is locked with an object. Then the game state model (obtained from the server as an XML structure and parsed into the PlayedGame property of the Account class) is traversed step by step and all required drawables are drawn as well. Each drawn object (player character, tower, library, summoner and army) has its own drawing method. However, all of them rely on a very similar concept. Let us examine more deeply just one of them, the DrawPlayer method

```
private void DrawPlayer(Canvas canvas, Player player)
{
    Bitmap playerBitmap = null;
    // If a player joins the game when it has already started, his drawable is not yet added in the
        respective
    // Bitmap Dictionary. To avoid a null exception, we have to add a new record into the
        dictionary.
    if (!PlayerBitmaps.TryGetValue(player.Name, out playerBitmap))
    {
        // The Colorize and GetPlayerDrawable methods are responsible for loading the respective
            bitmap and
        // add a team colour. They can be found in the appendix section.
        playerBitmap = Colorize(GetPlayerDrawable(player.Race), player.Colour);
        PlayerBitmaps.Add(player.Name, playerBitmap);
    }

    float centerComplement = ((float)Core.Global.FrameSize * Scale) / 2;
    float top = (((float)player.AnimationY + ResolveAnimationOffsetTop(player.AnimationOffset)) *
        Scale) - centerComplement;
    float left = (((float)player.AnimationX + ResolveAnimationOffsetLeft(player.AnimationOffset))
        * Scale) - centerComplement;
    float right = left + ((float)Core.Global.FrameSize * Scale);
    float bottom = top + ((float)Core.Global.FrameSize * Scale);

    Rect sourceRect = new Rect(Core.Global.AnimationPosition * Core.Global.FrameSize,
        (int)player.AnimationType * Core.Global.FrameSize,
        (Core.Global.AnimationPosition * Core.Global.FrameSize) + Core.Global.FrameSize,
        ((int)player.AnimationType * Core.Global.FrameSize) + Core.Global.FrameSize);

    canvas.DrawBitmap(playerBitmap,
    sourceRect,
    new RectF(left, top, right, bottom),
    null);
}
```

Code sample 9: DrawPlayer method

[66] http://www.slideshare.net/jserv/accel2drendering

The DrawPlayer method (called in the OnDraw method) takes as parameters the Canvas to draw to and the Player character that should be drawn. At the beginning the method tries to get the spritesheet Bitmap of a particular player. If none is found, a new Bitmap is created, loading the appropriate drawable with the GetPlayerDrawable method and then adding the team colour with the Colorize method.

After that, the method calculates several numeric values that are important during the drawing phase. Since all drawing starts at the top left corner, we need to adjust its value, so that the image looks centred. The centerComplement value serves for that purpose. It is calculated as the half of the global FrameSize constant (which is initialized in the LoginActivity as the product of the device's screen density and the number 96) and the WebView's Scale property (depending on the zoom level).

The top and left values are calculated from a client specific Player property, called AnimationX and AnimationY. When the Player is idle, the AnimationX property is the same as the Position.PosX property of the player's current location. However, during special events, like the movement animation, the AnimationX obtains custom values. Specifically, in the case of the movement animation, the AnimationX gets the interpolated values of the original and destination Location. The same applies to the AnimationY property. [67]

The next part of top (and left) values is obtained from the GetAnimationOffsetTop method. Since there can be several objects in the same Location (up to 5 players and 5 enemy armies, one tower or library or a summoner and a glyph), the drawables would overlap and hide each other. Therefore, in case multiple objects reside in the same location, their position is adjusted with this animation offset.

The sum of the AnimationX and the animation offset is multiplied by the zooming Scale of the WebView and the centerComplement is then subtracted from the resulting value. The bottom and the right values are then calculated from the top and left values, to which the global FrameSize multiplied with the Scale value are added.

All of these numbers served to specify only the destination rectangle that should be drawn. The source left coordinate is calculated from the global AnimationPosition value multiplied with the global FrameSize. The AnimationPosition value is a global value changed by the rendering timer thread. It specifies, which frame in the current animation should be displayed. When the timer advances, the AnimationPosition is incremented and the frame moves by one to the right.

```
private void _timer_Elapsed(object sender, System.Timers.ElapsedEventArgs e)
{
    Core.Global.AnimationPosition = (Core.Global.AnimationPosition + 1) % 4;
    Refresh(false);
}
```

Code sample 10: The change of AnimationPosition caused by elapsing timer

So to say, the AnimationPosition affects the horizontal movement of the frame in the spritesheet. The vertical movement is affected by the AnimationType global property, which - of course - selects the type of animation (idle, move left or right, attack left or right...). AnimationType multiplied with the FrameSize create the top coordinate of the

[67] The implementation of the animation can be found in the Appendix B.

source rectangle. The bottom and right coordinates are, again, calculated from the top and left coordinates by adding the FrameSize value.

At last, we have both the source and destination rectangles. Now, we can simply call Android's DrawBitmap method and pass it the selected player bitmap, the source and destination rectangles and a null value for the Paint.

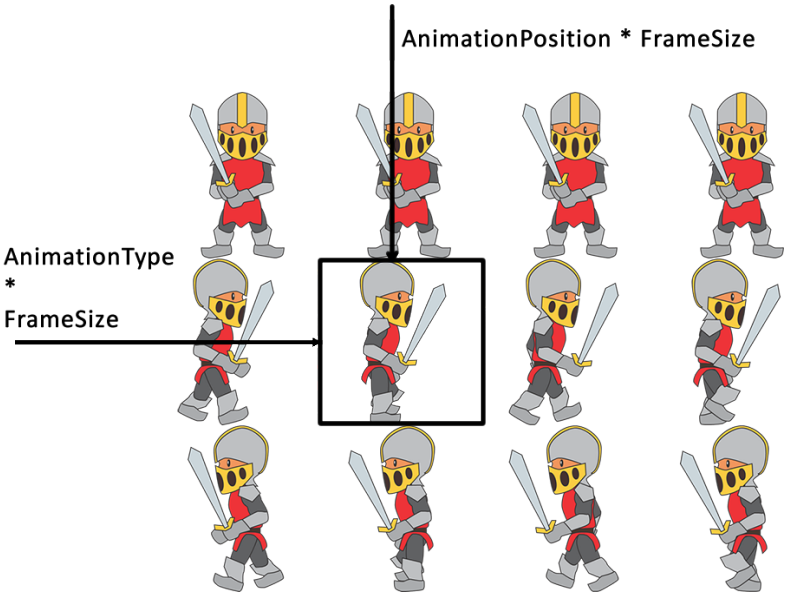

Figure 5: Movement animation frame selection

6.6.3 The Layout

The GameView of our GameTableActivity is finished, but that is only one part of the GameTable layout. The other part is a LinearLayout, which tracks the current statistics of the game, like score gained, the current day, turn, number of actions, remaining summoners, found glyphs, etc. This LinearLayout should be permanently visible on a tablet. However, on a smartphone, it should take the form of a slide-able side menu.

This approach is pretty similar to a very common Android design pattern, called the Navigation drawer [7]. The Navigation drawer pattern is very tightly coupled with Fragments. Clicking on an item in the menu provided by the Navigation drawer starts a new fragment in the content portion of the layout.

However, since our pseudo-drawer navigation serves only to display statistics, we do not need any fragments at all. We simply create a GameTable layout that consists of a MenuLayout with the statistics items and the ContentLayout with several buttons and out custom GameView. All we need now is to implement the sliding animation and toggling of the menu on a smartphone. Thankfully, there are several implementations for Xam-

arin.Android available. The simplest one is the Fading menu from QualTech [68]. However, the UI looks very poor and far from native. The most complex one is a Navigation drawer port made by a Xamarin member, Tomasz Cielecki [69]. However, since in Eridian the drawer is used in only a single Activity and it does not serve for navigation purposes, we can use the golden mean, a Fly-out menu by Cyril Mottier [70]. This implementation creates a custom class called FlyOutContainer, which has direct references to its children Views - MenuLayout and ContentLayout. It implements a nice toggling animation by simply changing its boolean AnimatedOpen property.

In case of tablets, another variation of the GameTable layout is used. Instead of the FlyoutContainer, the tablet GameTable layout displays a simple static layout with all elements (menu, buttons, GameView) always present. The screenshots from both phone and tablet versions can be found in the Appendix A.

6.6.4 Move

There are several ways a player can spend his in-game actions in order to save the world and win the game. Perhaps the most basic is moving around the game map. To do so, the player only needs to touch the screen on a place, where available movement destination is encircled. This means, our GameTableActivity has to implement the GestureDetector.IOnGestureListener interface.

When the OnSingleTapUp event is detected, GameTableView calls the Move(MotionEvent e) method. This method checks if the touch happened within a tolerated distance of a Location that is among available movement destinations. If so, a Move method is called on the SignalR server. The server processes the request, changes the Game state and broadcasts an XML response in the SendMove method to all players in the same game. The SendMove method is wired up with the OnMoved event in the client. The method subscribed to the OnMoved event and the complete sequence diagram can be found in the Appendix C. Both client and server methods involved in the moving process can be found in the Appendix B.

6.6.5 Other Actions

There are many more actions that a player can do during his turn. They include attacking enemy army, building a tower or library, cleansing, picking up and dropping glyphs and banishing enemy summoners. Each of them follows a similar SignalR communication pattern.

The player clicks on an actions button for the specific action (like attack). The client device checks, whether such action follows the local rules and then sends a request to the server. The server checks whether the request obeys the global rules (like interaction with other players) and then changes the game state. It also verifies the update index sent by

[68] http://www.qualtechsoftware.com/qualtechsoftware/2014/02/implementing-a-fadding-menu-android-with-xamarin/

[69] https://github.com/Cheesebaron/DrawerSample

[70] http://blog.neteril.org/blog/2013/04/19/fly-out-menu-xamarin-android/

the client with the update index stored in the Game. If those indices match, all players in the same game (SignalR group) are broadcasted only the partial update that is needed for displaying an animation and updating their local Game state copies.

However, in case that the indices do not match, the server knows there might have happened some connection error. Since the server does not keep a track of all the requests sent by the players, it cannot send just the needed updates. Instead, it sends the complete game state as an XML, broadcasting it to all players. This approach ensures that all users will receive the most recent update. However, it prevents the client from displaying appropriate animation, thus a lag is present.

6.7 Enemy Turn and End Game

There are many topics that have not been covered, but it is outside the scope of this thesis to examine them in detail. Just to list some of the omitted topics, here is a short (and non-exhaustive) list of them: Android development in Xamarin; mobile multiplatform development guidelines; threading, asynchrony and locking on the server Hubs and in client View rendering; specifics of SignalR programming; error handling in Xamarin, SignalR and asynchronous environment; Eridian implementation on iOS and Windows Phone; and many many others. Adding these topics would, however, stretch this already quite long thesis.

Thus, the only thing that remains to cover is the game's end. After a human player ends his turn, the enemy is on turn (in fact, the game system is). Depending on the game difficulty, a number of summoners summon an army in his reach. Let us look at the beauties of LINQ one last time:

```
public List<Location> SummonerReach
{
    get
    {
        List<Location> result = new List<Location>();
        result.AddRange(Position.Adjacent);

        if (Armies == null || Armies.Count <= 0) return result;
        Armies.ForEach(a => result.AddRange(a.Position.Adjacent));

        result = result.GroupBy(location => location.ID).Select(grouping => grouping.FirstOrDefault
            ()).ToList();
        result.RemoveAll(location => location.ID == Position.ID
            || location.Summoner != null
            || location.Adjacent.Any(a => a.Players.Any(p => p.Race == Enumerations.RaceType.
                Human)));
        return result;
    }
}
```

Code sample 11: Summoner reach - complex LINQ example

SummonerReach is a List of Locations. All Location adjacent to the summoner himself belong to that List. Also, all Location that are adjacent to any army of the summoner's

race belong to that List. In the end, we select only distinct locations and remove all those that are adjacent to a player that has the human crusader character. This is related to crusader's special ability - he is able to protect all adjacent locations from spreading of enemy armies. Also, the other characters have special abilities. Elven ranger is able to attack not only the location she is standing on, but also adjacent locations. Orc berserker can kill two enemy units consuming only one action point. The undead necromancer is able to pick up tainted glyphs, cleansing them automatically for free. Other characters have to spend all their remaining actions for cleansing a glyph. And the vodnik oracle is able to see glyphs also on adjacent locations, while the other characters can only see a glyph on the location they are standing on.

Now, back to spreading of enemy armies. Armies of the same race can stack up in a location. Whenever an army stack reaches 3 units and a 4th unit should be added, a terrible event happens instead. A terrible event is a period of game (usually one day), during which the game rules are slightly altered. Players have less action points, summoners summon two units instead of one, the special abilities of players do not work, etc.

If the players manage to find all glyphs and bring them to their capitals or libraries, they get the titan blessing and are able to banish the summoners. When all summoners are banished, players win the game. However, if the players do not manage to do so within 30 days, they lose the game. The same happens in case there were any 3 enemy army units in any capital.

Upon reaching the victory or defeat condition, the server sends the GameResultSend message containing the last valid Game state and the result of the game. When client devices get the server message, the OnGameResultUpdate event is fired. This tells the GameTableActivity that it should immediately start the ResultActivity. The ResultActivity contains the result of the game, a score board and a button OK that sends the player back to the LobbyActivity for one more game.

7 Implementation Problems

Although we have finished a walk through all activities, there are still some topics that we did not cover. They are related to issues that emerged during the implementation phase, but had no direct or exclusive connection to any activity.

7.1 Sharp Images in WebView

When displaying a large image in a WebView, you will most probably encounter a problem. The image will look blurry, pixelated and in low quality, although the original resource may have been crispy sharp. This is a "desired" behaviour of Android, since it tries to save resources.

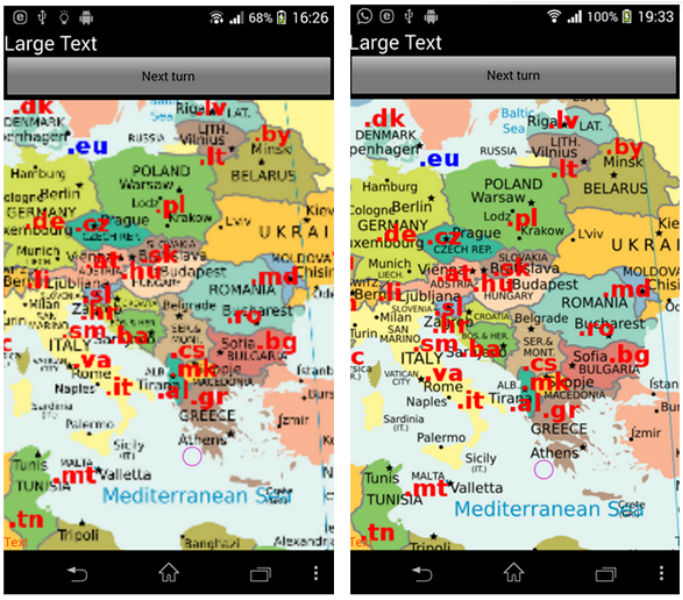

Figure 6: Comparison of default and adjusted image sharpness

As always, there are several solutions and workarounds. The first of them is to use a picture with higher density (DPI). However, this approach solves the problem only in case zooming is not enabled. Whenever the user zooms the image, the higher DPI will sooner or later lose its effect and the image will look blurry anyway. Moreover, high-density images take up much more memory.

Another approach that will work also when zooming, is loading images from the appropriate Android drawable folder and then recounting the density manually[71]. This might have been the solution for an Java Android application, but Xamarin has a limitation here. It is unable to address the drawable directory from a stream. Only the assets

[71] http://stackoverflow.com/a/22079820/1942656

folder can be accessed. However, assets are not grouped by device screen density, thus this solution is unavailable for Xamarin Android applications.

The last solution may at first seem as the most toilsome, but it brings the best (and most juicy) fruit. Since the WebView displays an HTML page, we can simply chop the large image into small pieces and then bring them back together in a HTML table. This approach was used also in the Eridian game and, as you can see, the result is more than satisfyingA.

7.2 Strange SignalR Behaviour in Xamarin.Android

Since both Xamarin.Android and SignalR are quite young technologies, they are not yet completely bug free. Although they are becoming better and better with each release, there can still be encountered some paranormal activity that cannot be explained rationally. It is unclear, whether the problem is in the Xamarin implementation, the SignalR framework or simply in their communication.

```
_proxy.On("SendGameJoined", (object xDoc) =>
{
    if (OnGameJoined != null) OnGameJoined(this, xDoc);
});
```

Code sample 12: Xamarin SignalR compilation bug - object type

The first example is the wiring up of a client method called by the server, which should fire an event. When compiling the project, the build failed providing the following error message: System.MissingMethodException: Method not found: 'Microsoft.AspNet.SignalR .Client.HubProxyExtensions.On'. The strange thing is that, when the object parameter was replaced by a string, everything went smoothly. The same error happened when trying to compile a method with 5 parameters:

```
_proxy.On("SendMove", (string playerID, string fromID, string toID, int actions, int lastUpdate)
    =>
{
    if (OnMoved != null) OnMoved(playerID, actions, fromID, toID, lastUpdate);
});
```

Code sample 13: Xamarin SignalR compilation bug - 5 parameters

Again a very strange behaviour, considering that SignalR should be able to support up to 7 parameters of any type. Thankfully, there was a simple workaround in this case as well. Since the ID of the Location from which a Player was moving is known in the client, it could be removed. The same method with just 4 parameters compiled successfully.

```
_proxy.On("SendMove", (string playerID, int actions, string destinationID, int lastUpdate) =>
{
    if (OnMoved != null) OnMoved(playerID, actions, destinationID, lastUpdate);
});
```

Code sample 14: Xamarin SignalR compilation bug - workaround

So far, neither a systematic solution nor explanation of the behaviour was discovered. If the reader was to know the possible reason of these compilation errors, he/she is most welcome to answer the question at StackOverflow [72].

7.3 Double Click on Samsung Tablets

Eridian was tested on various emulators and physical devices. The game was subjected not only to visual tests, but also functional tests (test result details can be found in the chapter 8). Strange behaviour was detected on Samsung Galaxy Tab S 10.5" SM-T805. In the GameTableActivity, when a user clicked on the GameView, a double click was performed instead. This caused unwanted behaviour. Although the Player moved to an adjacent location, he lost 2 action points. Occasionally, when the player had last remaining action point and performed a movement, he resulted in having negative action points. Several error messages were displayed as well, informing him that he has no more action points, he is not on his turn or it is not possible to reach the desired destination.

The author was unable to detect the source of this strange behaviour or a solution to it. However, similar behaviour was reported by many other programmers using Xamarin, PhoneGap, Sencha and other platforms [73][74][75]. Since double tap has no functionality in Eridian game, a simple work-around quickly offered itself.

The GameTableActivity was extended to implement GestureDetector.IOnDoubleTapListener . The Move method call was removed from the OnSingleTapUp event. Instead, OnSingleTapConfirmed and OnDoubleTap methods were both implemented, both calling the same Move function. This work-around proved to solve the problem. However, in the case when double tap had a different functionality from single tap gesture, this Samsung-relate bug would cause serious trouble and a possible work-around would be much more complicated.

[72] http://stackoverflow.com/questions/27669362/possible-signalr-bug-in-xamarin-android

[73] http://forums.xamarin.com/discussion/comment/110868/#Comment_110868

[74] https://github.com/jquery/jquery-mobile/issues/3340

[75] http://www.sencha.com/forum/showthread.php?230646-tap-event-fires-two-events-on-android

8 Testing

As soon as the Eridian game was finished, it became the subject of several tests. A small group of non-professional testers have assessed the game-play, rules, balancing and overall user experience on several devices. A series of technical tests was performed by the author as well. Their results are summarized in the following chapters.

8.1 Testing Screen Sizes and Resolution

The final implementation was tested on several physical and emulated devices. Minimal requirements include Android version 4.0 or higher (API 14). The game can be played on any screen size with any resolution. However, at least WQVGA400 (240 x 400 px, 120 dpi) or WVGA800 (480 x 800 px, 160 dpi) screens are advised. The game is playable also on smaller screens (QVGA and HVGA), however some parts of the GUI may not be rendered properly (screenshot in Appendix A).

The list of tested physical devices:

- Google Nexus 7 - Android 5.1
- HTC Desire 610 - Android 4.4.2
- HTC One V - Android 4.0.3
- Samsung Galaxy Tab S10.5" SM-T805 - Android 4.4.2
- Samsung Galaxy Tab 2 - GT P5100 - Android 4.0.4
- Sony Xperia C5303 - Android 4.3
- Sony Xperia P - Android 4.0.4

Eridian game was tested also on various virtual devices, emulated by Android SDK emulator, Genymotion emulator and Xamarin Android Player. Regarding the graphical user interface, no issues were detected. However, there was a strange bug with the Samsung Galaxy tablets (as described in subsection 7.3). Several in-game screenshots can be seen in appendix A.

8.2 Testing Low Connection Speeds

Since the Internet connection conditions are not always ideal, Eridian game was subjected to data transportation tests as well. The tests were performed on a emulated device which communicated with real server. The Android SDK emulator enables simulating various conditions. After a telnet connection is established with the emulated device, the network speed can be slowed down with the `network speed [SPEED]`.

Eridian was tested with all available options. The speed values of two main messages were tested. Small updates, which consisted of a pair of IDs and strings. These messages have around 100 bytes (pure message body, without SignalR and HTTP headers). Large

Network speed	Small message (100 bytes)	Large message (10 kB)
GSM	565.92 ms	9996.5 ms
HSCSD	464.07 ms	17681.25 ms
GPRS	399.55 ms	4975.93 ms
EDGE	433 ms	1975.75 ms
UMTS	386.83 ms	1758.38 ms
HSDPA	343.37 ms	1011.8 ms
Full speed	299.85 ms	893.25 ms

Table 1: Network transportation speed comparison

updates contained the whole XML structure of the game and had approximately 10 - 12 kB. To create similar conditions, always the same game type and number of players were selected. However, possible fluctuations in real network speed cannot be excluded.

The time interval for each message was measured on the client device using the System.Diagnostics.Stopwatch class available in Mono .NET. The stopwatch were started just before a request was sent to the server and were stopped right after a response was received. The average speed values of individual network settings for both small and large messages can be seen in a table below.

Several interesting points can be observed in the table. While the transportation speed of large chunks of data with no restrictions is more than 10-times faster than on GSM, small data are transported only by 48% faster. Although the overall trend is that with each newer technology the transportation speed is faster, there are two exceptions. First of them is HSCSD connection, in which the transportation of large pieces of data is almost twice slower than in the GSM technology. The second exception is EDGE, which speeded up large data transportation by 61%, the small data transportation speed was slowed down by 8%.

From these data we can assume that the game is playable from connection speeds of the same level as EDGE (with occasional small lags during large updates). However, for ideal user experience, at least HSPDA transportation is recommended.

8.3 Downgrading to Lower Versions of Android

Eridian game client application has been originally developed for Android 4.0 - Ice Cream Sandwich (API 14) and higher. However, several tests were performed to investigate whether previous versions of Android could be supported as well.

Xamarin supports all Android releases beginning with Android 2.1 (API 7). Downgrading the application to this version, however, broke the whole user interface and caused several chained compilation errors. Multiple changes would have to be performed on the application, therefore this configuration was not tested.

Similar to API 7, also on API 10 (Android 2.3 Gingerbread) was the application not able to compile. Although the user interface caused no more errors, several assembly references and method calls could not be resolved (because they were added in later

versions of Android). The compilation errors were caused (among others) by WebView(Context context, global::Android.Util. IAttributeSet attrs , **int** defStyle , **bool** privateBrowsing) constructor, BitmapFactory.Options.InMutable property, LinearLayout.GetX() method and the whole Android.Animation library. After removing or rewriting these problematic parts of code, the application was able to compile and run. However, the program aborted immediately after starting the GameTableActivity, since immutable bitmaps were passed to the Colorize method (Appendix B), which tried to repaint them.

Android 3.1 Honeycomb (or API 12) already produced a complete .APK file without any compilation errors. The GUI displayed properly, without glitches and the game was playable and went smooth. There were only occasional issues with displaying status messages (Eridian uses custom implementation of Android AlertDialog). However, it can be assumed that after thorough optimization and bug removal, the Eridian game could be fully playable also on Android Honeycomb devices.

In order to make the tests complete, Android 5.0 Lollipop (API level 21) was tested as well. As expected, the application was compiled without any problems. The Lollipop version of Eridian was tested only on a single device (Google Nexus 7). However, according to the owner of the device (a volunteer game tester) the Lollipop version of Eridian ran smoother and faster compared to the original Ice Cream Sandwich version.

8.4 Testing HTTPS Transport

Since SignalR framework communicates via the HTTP protocol, we have tested also the possibility of using SSL secured connection. Due to the fact that the game server is hosted on a school server, the author had only limited access to its administration. Thus, the HTTPS connection was tested only on localhost. Also, to be precise, Eridian game itself was not tested. Instead, a simple test application was created. If the reader was interested, he can reproduce the tests. A simple guide on how to set up local IIS to support HTTPS can be found in the Appendix D. It should be noted that since SignalR is decoupled from IIS and follows the OWIN specification, both the Eridian game server and the HTTPS test server can be run on Apache with Mono runtime, as well [1].

On the server side, no special changes need to be made. The SignalR Hub is creates precisely the same way as when running on unsecured HTTP protocol. However, to force the server to run under HTTPS, a small change needs to be made in the project properties (described in the Appendix D).

The client test application consists of a singular Android activity. It contains two inputs, two buttons and a text field. The purpose of the first input and first button is to create a new connection to the SSL SignalR server. As soon as the user clicks on the "Set new IP" button, a request is sent to the SignalR server. However, the server responds with a certificate that is unknown to the Mono runtime.

A new ServicePointManager.ServerCertificateValidationCallback has to be implemented to give the user the possibility to choose whether to accept the certificate or refuse it. This is achieved by the Android AlertDialog. The problem with Android dialogues is that they are run on a separate thread. Thus, the ServicePointManager.ServerCertificateValidationCallback

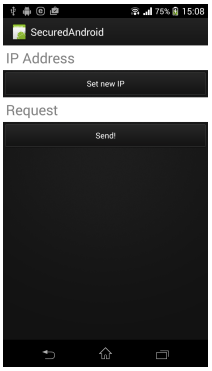

Figure 7: SSL client UI

will return false and Mono runtime will return an error message, informing about invalid server certificate. Thus, no connection will be established.

Since ServicePointManager.ServerCertificateValidationCallback is a callback function running in the main application thread and triggered by the Mono runtime, and the result of an AlertDialog is a callback function running in the UI thread and triggered by the user, there is no easy communication between these two methods. One approach, how to resolve this situation is to simply ignore the first unsuccessful attempt to connect to the server. We wait until the user decides, whether he trusts the certificate. Then we use his decision as a new ServicePointManager.ServerCertificateValidationCallback result and call the connection start method again. If the user decided to accept the certificate, an SSL encrypted connection is established and the full potential of the SignalR framework is at our disposal.

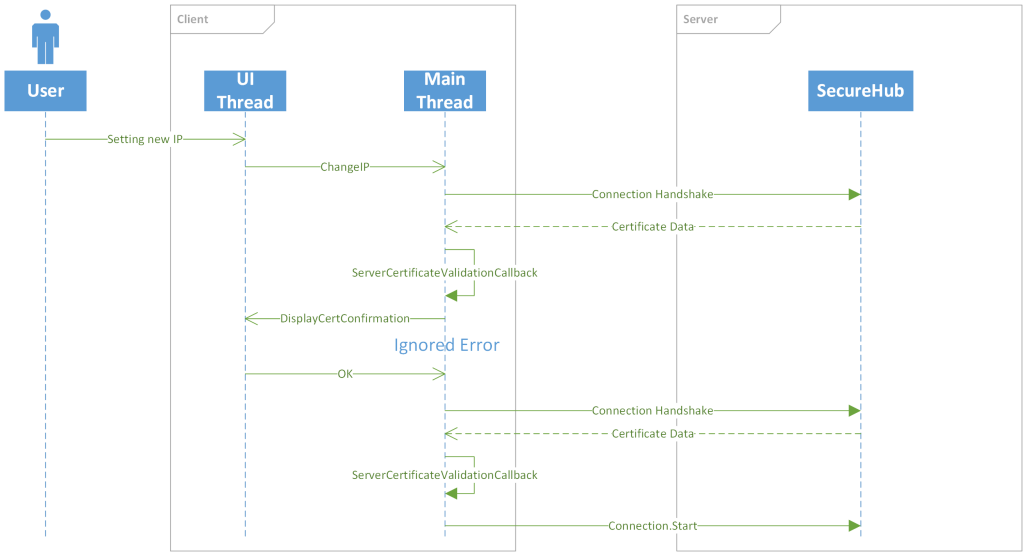

Figure 8: SSL connection handshake and certificate approval

9 Conclusion

In this thesis, we have successfully managed to design and implement a new board game for mobile devices running on the Android platform. At first, we have examined existing board games and synthesized the best of them into a brand new set of rules. After that, we have researched the mobile market and chosen Android platform as the primary one, but leaving open door for iOS and Windows Phone platforms. To prevent rewriting the code in three different programming languages for three different platforms, we have examined the possibilities of multiplatform development and chosen the Xamarin framework. The usage of .NET platform and the promising PUSH Internet communication model have lead us to investigate SignalR libraries.

Because of using new and unconventional technologies, we had to use unorthodox design patterns in the implementation phase as well. Although we faced several compilation errors and application bugs, we were able to solve (or at least work around) all of them. As a result, we have v created a functional board game that has passed several performance tests.

The main contribution of this thesis can be seen in the research of multiplatform development of 2D multiplayer mobile games without the use of game frameworks and engines. The thesis has also investigated the methods and design patterns for modern .NET technologies: SignalR client-server communication and Xamarin mobile development.

There is certainly a lot of space for further development. As stated above, the game is ready to expand on iOS and Windows Phone platforms. The possibility to establish peer-to-peer connection could be added as well. The game itself could be enriched by new features and integrate in-game chat or a VOIP service. The game could also be converted into 3D, however this would require complex redevelopment.

10 References

[1] AGUILAR, José M. *SignalR programming in Microsoft ASP.NET.* xix, 243 pages. ISBN 07-356-8388-3.

[2] BERLAND, Matthew a Victor R. LEE. Collaborative Strategic Board Games as a Site for Distributed Computational Thinking. *International Journal of Game-Based Learning* [online]. 2011, vol. 1, issue 2, s. 65-81 [cit. 2015-04-04]. DOI: 10.4018/ijgbl.2011040105.

[3] CHAPPELL, Laura, CARRELL, Jeffrey L, TITTEL, Ed and PYLES, James. *Guide to TCP/IP.* Fourth edition /. xxii, 744 pages. ISBN 11-330-1986-2.

[4] CHO, James S. *The Beginner's Guide to Android Game Development.* Java + Android edition. Glasnevin Publishing , 2014, 438 pages. ISBN 978-1-908689-26-9.

[5] LÖWY, Juval. *Programming WCF services.* 3rd ed. Sebastopol: O'Reilly, 2010, xxx, 875 s. ISBN 978-0-596-80548-7.

[6] MEIER, Reto. *Professional Android 4 application development.* Updated for Android 4. Indianapolis: John Wiley & Sons, 2012, xlii, 817 p. ISBN 978-111-8262-153.

[7] NUDELMAN, Greg. *Android design patterns: interaction design solutions for developers.* 1st ed. Indianapolis, Ind.: Wiley, 2013, xxvii, 423 p. ISBN 978-1118394151.

[8] PHILLIPS, Bill and HARDY, Brian. *Android programming: the Big Nerd Ranch guide.* 1st ed. xxii, 602 pages. ISBN 03-218-0433-3.

[9] REYNOLDS, Mark. *Xamarin Mobile Application Development for Android.* Birmingham: Packt Publishing, 2014. ISBN 978-178-3559-169.

[10] SHACKLES, Greg. *Mobile development with C#.* 1st ed. Sebastopol, CA: O'Reilly, c2012, xi, 155 p. ISBN 14-493-2023-6.

[11] SILVA, Vladimir. *Pro Android games.* Second edition. xix, 392 pages. ISBN 978-1430247975.

[12] TROELSEN, Andrew W. *Pro C# 2010 and the .NET 4 platform.* 5th ed. New York, NY: Apress, c2010, xxxviii, 1712 p. ISBN 978-1-4302-2549-2.

[13] ZECHNER, Mario and GREEN, Robert. *Beginning Android 4 games development.* New York: Distributed to the book trade by Springer, c2011, xv, 677 p. ISBN 978-1-4302-3987-1.

A Eridian Game Screenshots

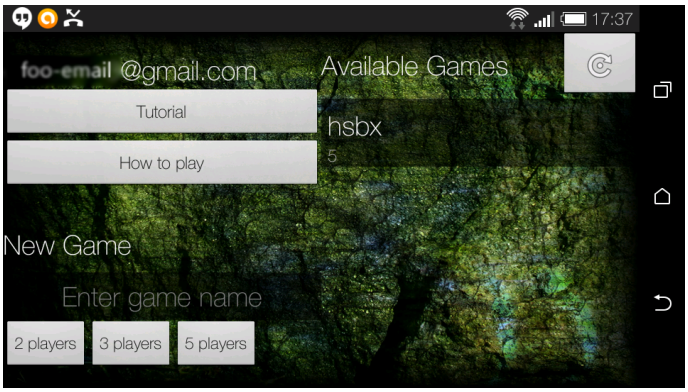

Figure 9: Lobby screenshot on a smartphone (4.7 in)

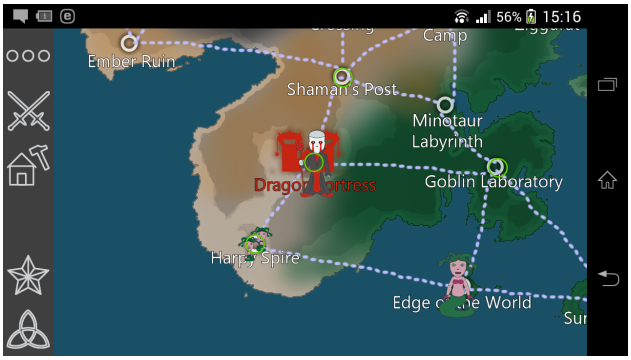

Figure 10: Gameplay screenshot on a smartphone (4.6 in)

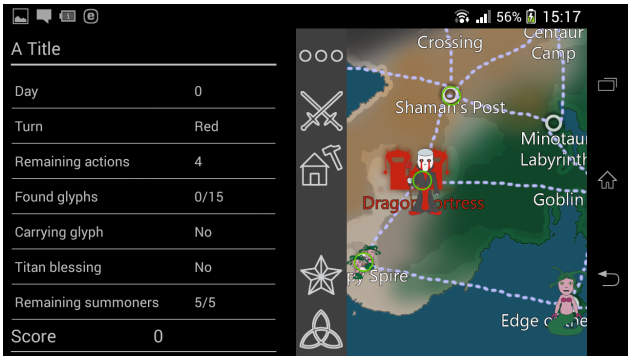

Figure 11: Gameplay screenshot on a smartphone with menu (4.6 in)

Figure 12: Gameplay screenshot on a tablet (10.5 in)

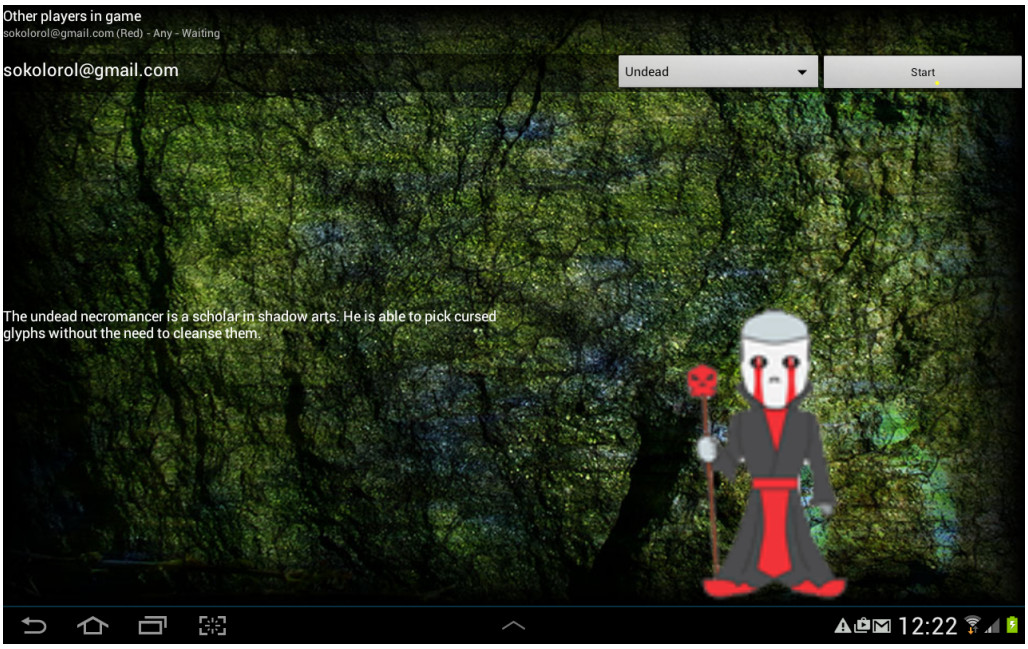

Figure 13: Character choice screenshot on a tablet (10.1 in)

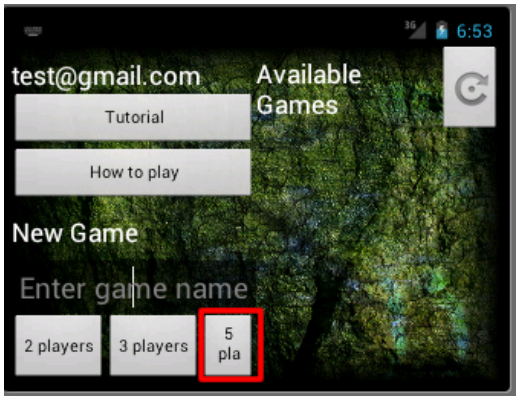

Figure 14: Lobby screenshot in QVGA emulator

Figure 15: Gameplay screenshot in QVGA emulator

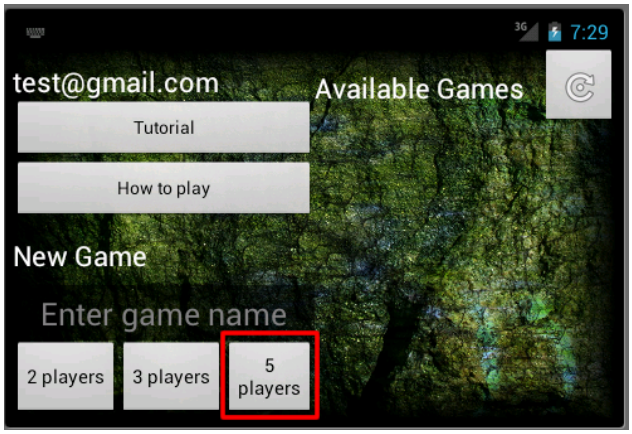

Figure 16: Lobby screenshot in HVGA emulator

B Code Sample Digest

```
private Bitmap Colorize(Bitmap mutable, Core.Enumerations.PlayerColour playerColour)
{
    if (mutable == null) return null;

    Canvas mutableCanvas = new Canvas(mutable);

    mutableCanvas = Colorize40White(mutableCanvas, playerColour);

    Paint paint = new Paint();
    paint.SetARGB(255, AL.Constants.TeamReplacementColour.R, AL.Constants.
        TeamReplacementColour.G, AL.Constants.TeamReplacementColour.B);
    paint.AntiAlias = true;
    Color removeColour = paint.Color;

    Color racialColour = ResolveColour(playerColour);

    paint.SetARGB(255, racialColour.R, racialColour.G, racialColour.B);
    paint.SetXfermode(new AvoidXfermode(removeColour, 25, AvoidXfermode.Mode.Target));

    mutableCanvas.DrawPaint(paint);
    paint.Dispose();
    mutableCanvas.Dispose();
    return mutable;
}
```

Code sample 15: Adding team colour

```
private void Client_OnMoved(string playerID, int actions, string toID, int lastUpdate)
{
    ...
    ValueAnimator animator = ValueAnimator.OfObject(new Utils.LocationEvaluator(),
    new Point(from.PosX, from.PosY, null),
    new Point(to.PosX, to.PosY, null));

    animator.SetInterpolator(new global::Android.Views.Animations.LinearInterpolator());
    animator.SetDuration(1000);

    animator.Update += (sender, e) =>
    {
        Point newPoint = (Point)e.Animation.AnimatedValue;
        if (newPoint == null) return;

        player.AnimationX = newPoint.X;
        player.AnimationY = newPoint.Y;

        Refresh(false);
    };

    animator.AnimationEnd += (sender, e) =>
    {
        player.AnimationX = null;
        player.AnimationY = null;
        player.AnimationType = Core.Enumerations.PlayerAnimationType.Idle;
        player.PositionID = to.ID;
        player.Actions = actions;

        animator.RemoveAllListeners();
        animator = null;

        InputAllowed = true;
        Refresh(true);
    };

    InputAllowed = false;
    RunOnUiThread(() =>
    {
        animator.Start();
    });
}
```

Code sample 16: OnMoved method

```
protected async void Move(MotionEvent e)
{
    if (!InputAllowed || DialogueDisplayed) return;
    if (PersistentContext.LoggedPlayer == null
        || PersistentContext.LoggedPlayer.CurrentPosition == null
        || PersistentContext.LoggedPlayer.CurrentPosition.Adjacent == null) return;

    int [] viewPosition = new int[2];
    viewMain.GetLocationOnScreen(viewPosition);

    if (e.RawX < (llSideBar.GetX() + (float)llSideBar.Width)) return;
    float StartX = (e.RawX - viewPosition[0] + (float)viewMain.ScrollX) / viewMain.Scale;
    float StartY = (e.RawY - viewPosition[1] + (float)viewMain.ScrollY) / viewMain.Scale;

    Core.DAL.Location destination = PersistentContext.LoggedPlayer.MovableLocations.
        MovementDestination(StartX, StartY, viewMain.Scale);

    if (destination != null)
    {
        await AL.Connection.Client.Move(AL.Constants.AsyncDelay, PersistentContext.LoggedPlayer.
            Name, PersistentContext.PlayedGame.ID, destination.ID);
    }
}
```

Code sample 17: Client Move method

```
public static Location MovementDestination(this List<Location> map, float desiredX, float
    desiredY, float scale)
{
    Location destination = null;
    if (map == null) return null;
    float tolerance = Global.TouchTolerance * scale;

    destination = map.FirstOrDefault(loc
                => loc != null
                && desiredX > (loc.PosX - tolerance)
                && desiredX < (loc.PosX + tolerance)
                && desiredY > (loc.PosY - tolerance)
                && desiredY < (loc.PosY + tolerance));

    return destination;
}
```

Code sample 18: Client extension method adding touch tolerance

```
public void Move(string playerName, string gameID, string destinationID, int lastUpdate)
{
    Game game = null;

    if (!PersistentContext.Games.TryGetValue(gameID, out game))
    {
        var error = Xml.MakeError("Game not found.");
        Clients.Caller.SendGameViewUpdate(error.OuterXml);
        return;
    }

    Player player = game.Players.Where(item => item.Name == playerName
        && item.Name == Context.Headers["userName"].ToString()
        && item.Token).FirstOrDefault();

    ...

    Location previousLocation = player.Position;

    player.PositionID = destination.ID;
    player.Actions--;
    PersistentContext.Players.AddOrUpdate(playerName, player, (s, p) => player);

    if (lastUpdate != game.LastUpdateIndex)
    {
        game.LastUpdateIndex++;
        game.SerializedGame = game.XGame().ToString();
        foreach(Player user in game.Players)
        {
            Clients.User(user.Name).SendGameViewUpdate(game.SerializedGame);
        }
    }
    else
    {
        game.LastUpdateIndex++;
        game.SerializedGame = game.XGame().ToString();

        foreach(Player user in game.Players)
        {
            Clients.User(user.Name).SendMove(player.Name,
            player.Actions,
            destination.ID.ToString(),
            game.LastUpdateIndex);
        }
    }

    if (player.Actions <= 0)
    {
        ResolvePlayer0Actions(player, game);
    }
}
```

Code sample 19: Server Move method broadcasted to whole game room

```
public void StartNewGame(string playerName, string gameName, int maxPlayers)
{
    Game game = PersistentContext.Games.FirstOrDefault(g => g.Value.Name == gameName).
        Value;

    if (game != null)
    {
        var error = Xml.MakeError("Game␣with␣that␣name␣already␣exists.");
        Clients.User(Context.Headers["userName"]).SendGameJoined(error.OuterXml);
        return;
    }

    Player player = null;
    bool alreadyPlays = PersistentContext.Players.TryGetValue(playerName, out player);

    if (alreadyPlays)
    {
        lock(oldGamesLock)
        {
            HubHelper.RemoveFromOldGames(player);
        }
    }

    game = new Game(gameName, maxPlayers);
    game.Init();

    player = new Player()
    {
        Actions = 4,
        Colour = HubHelper.ResolvePlayerColour(game),
        ConfirmedGame = false,
        ConnectedOn = DateTime.Now,
        GameID = game.ID,
        IsTitan = false,
        Name = playerName,
        Race = Enumerations.RaceType.Any,
        Score = 0,
        Token = HubHelper.ResolveInitialToken(game)
    };

    PersistentContext.Players.AddOrUpdate(playerName, player, (s, p) => player);

    game.SerializedGame = game.XGame().ToString();
    Clients.Caller.SendGameJoined(game.SerializedGame);
}
```

Code sample 20: Server method for starting new game

C UML Diagram Digest

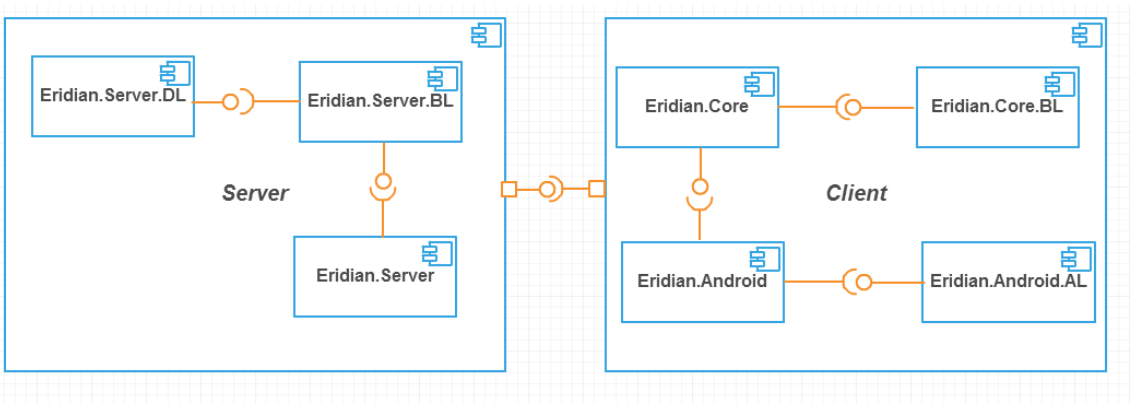

Figure 17: Deployment Diagram

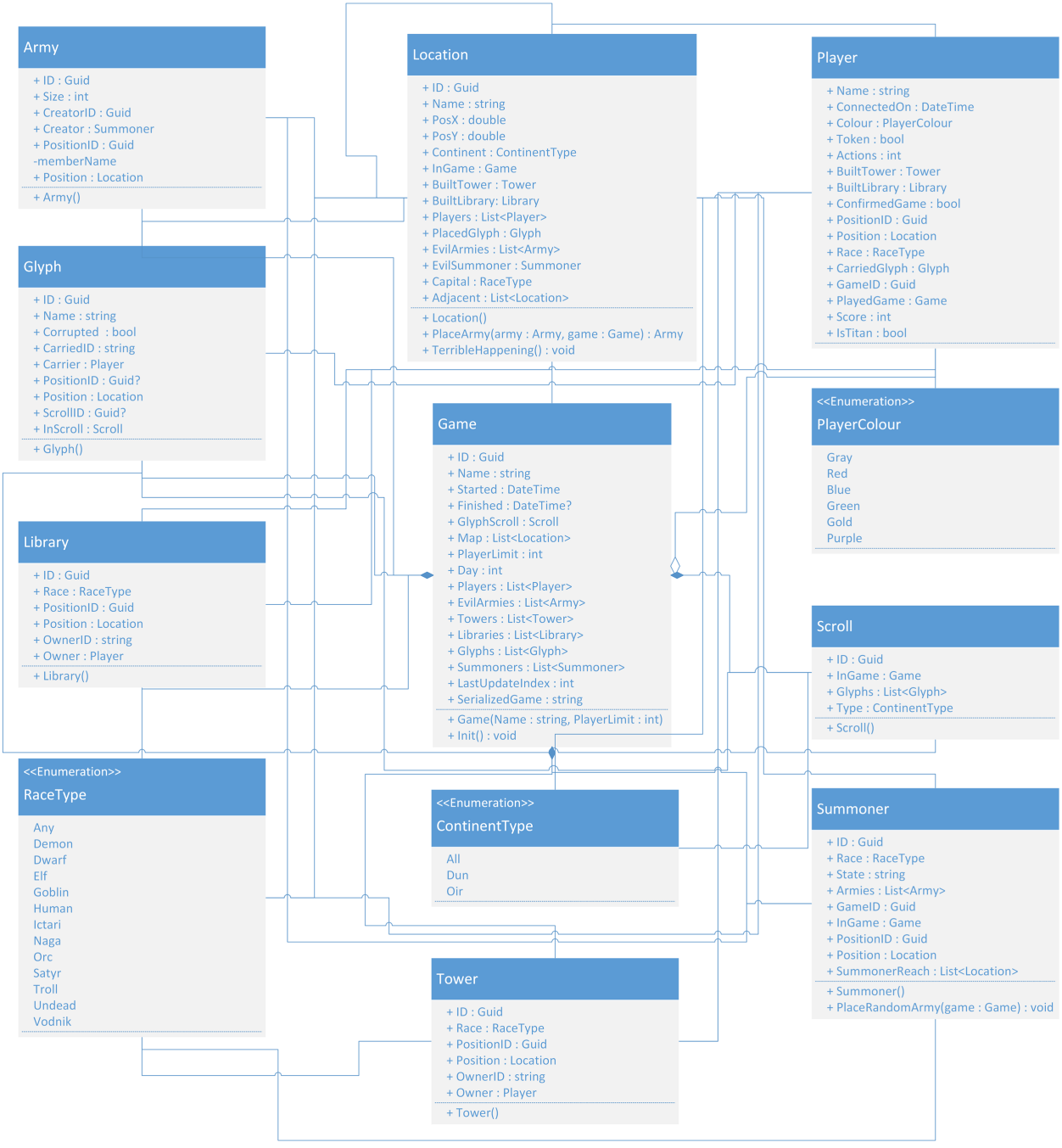

Figure 18: Complex business layer class diagram

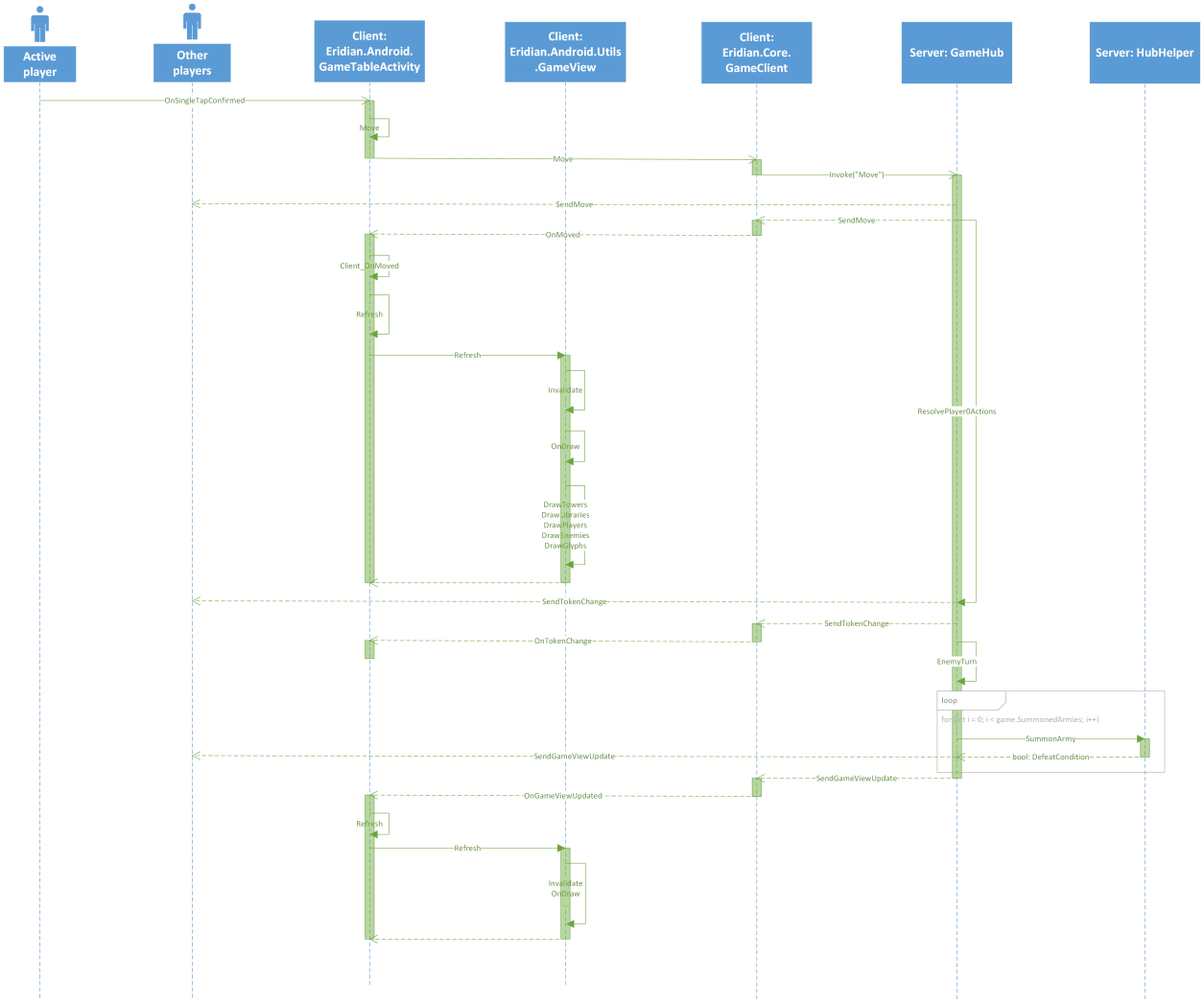

Figure 19: Complex sequence diagram of movement action

D Setting Up HTTPS on IIS Localhost

The following steps serve as a guide for setting up SSL connection in local IIS. The guide was written for IIS 7 and 7.5. There may be small differences when creating a SSL connection on different versions of IIS:

1. Open Internet Information Services (IIS) Manager.
2. Click on the **Server Certificates** icon in the main panel.
3. Click on the **Create Self-Signed Certificate...** button in the right Actions pane.
4. Name your certificate and click on OK.
5. Next, we need to make sure the certificate is among Trusted Root Certification. Press Windows key + 'R' to open the Run command. Type 'mmc' and click OK.
6. Click on the File menu and select **Add/Remove snap-in...**
7. Select **Certificates** and click on Add. From the options available, choose **Computer account** and then **Local computer**. Click OK.
8. Open the Certificates list in the left pane and choose **Personal** > **Certificates**.
9. Here, you should find your certificate. Copy it. Then, navigate to **Trusted Root Certification Authorities** > **Certifica**tes. Paste the copied certificate here.
10. Now, back to IIS. Click on the Web site (in Sites folder in left pane) where you want to host the SSL server.
11. Click on **Bindings...** in the right Actions pane.
12. Select Add binding. Choose type HTTPS. The default secure port, 443, can be used. Choose your SSL certificate and click OK.
13. The IIS is now successfully set to support SSL connection. To force the C# server to run under HTTPS, certain configuration changes need to be done to the server project as well. Open the solution in Visual Studio and open project properties (F4).
14. Click on the **Web** button. In the Servers section, Local IIS as the host. Specify the **Project Url**, ensuring that HTTPS protocol is used.
15. As a last step, click on the **Create Virtual Directory** button. This has to be done only when the server solution is run for the first time.

Now, we can connect to the SignalR server. However, only requests from localhost will be accepted. To connect from external devices (within the same local network), a firewall rule has to be created to allow connection to the running server.

SignalR hubs can be hosted also on Apache servers with installed Mono runtime. Note that SignalR applications can be also self-hosted. Assigning the certificate to certain port can be accomplished by the following console command:

```
netsh http  add sslcert  ipport=<ip address>:<port> appid={12345678-db90-4b66-8b01-88
    f7af2e36bf} certhash=<certificate hash>
```

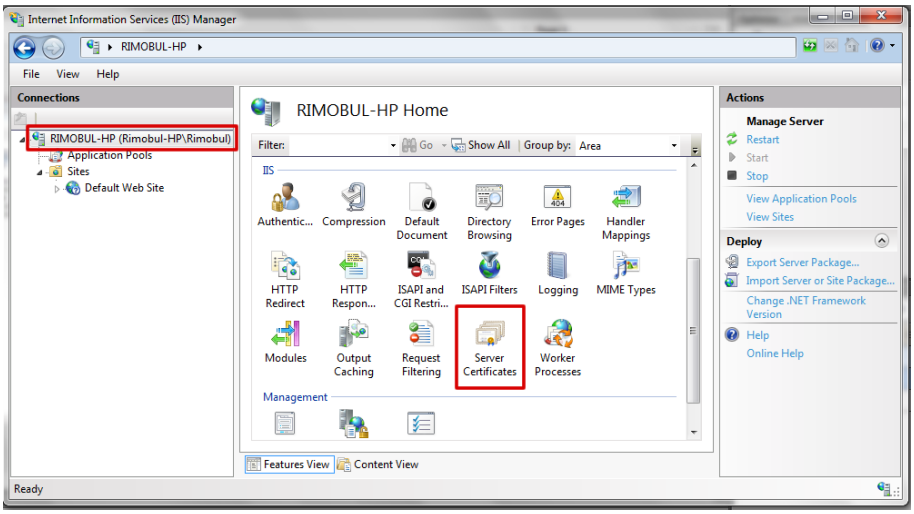

Figure 20: IIS set up - Server Certificates

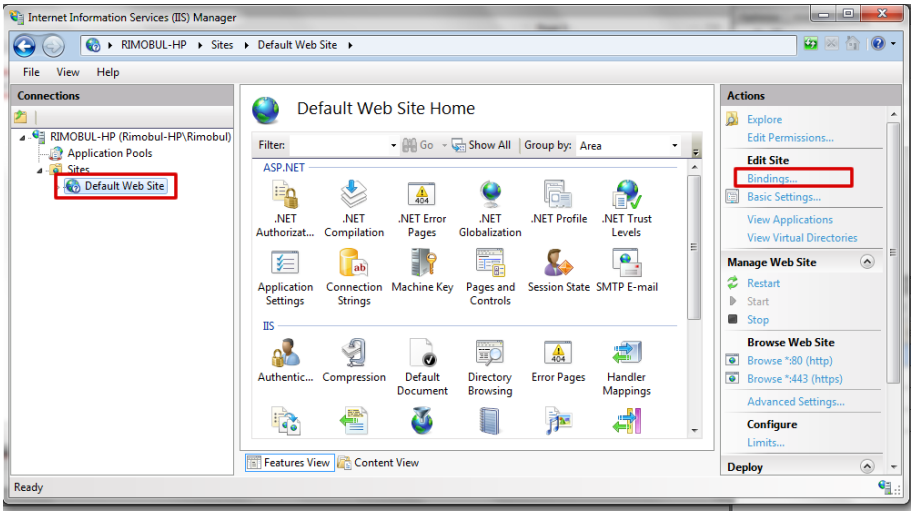

Figure 21: IIS set up - Web site Bindings

E CD Contents

1. Eridian game solution (Visual Studio project)
2. Eridian game server deployment package with client installation package (.apk)
3. Eridian game server solution (Visual Studio project)
4. SecuredAndroid test client application (Visual Studio project)
5. SecuredSignalR test server (Visual Studio project)
6. This document in .pdf format